Snapshots in Large-Scale Distributed File Systems

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Dipl.-Inf. Jan Stender

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Alexander Reinefeld
2. Prof. Dr. Miroslaw Malek
3. Prof. Dr. Guillaume Pierre

eingereicht am: 14. September 2012
Tag der mündlichen Prüfung: 4. Januar 2013

Abstract

Snapshots are present in many modern file systems, where they allow to create consistent on-line backups, to roll back corruptions or inadvertent changes of files, and to keep a record of changes to files and directories. While most previous work on file system snapshots refers to local file systems, modern trends like cloud and cluster computing have shifted the focus towards distributed storage infrastructures. Such infrastructures often comprise large numbers of storage servers, which presents particular challenges in terms of scalability, availability and failure tolerance.

This thesis describes a snapshot algorithm for large-scale distributed file systems and its integration in XtreemFS, a scalable object-based file system for grid and cloud computing environments. The two building blocks of the algorithm are a version management scheme, which efficiently records versions of file content and metadata, as well as a scalable and failure-tolerant mechanism that aggregates specific versions in a snapshot. To overcome the lack of a global time in a distributed system, the algorithm implements a relaxed consistency model for snapshots, which is based on timestamps assigned by loosely synchronized server clocks. More precisely, the algorithm relaxes the point in time at which all servers capture their local states to a short time span that depends on the clock drift between servers. Furthermore, it ensures that snapshots preserve the causal ordering of changes to files and directories.

The main contributions of the thesis are: 1) a formal model of snapshots and snapshot consistency in distributed file systems; 2) the description of efficient schemes for the management of metadata and file content versions in object-based file systems; 3) the formal presentation of a scalable, fault-tolerant snapshot algorithm for large-scale object-based file systems; 4) a detailed description of the implementation of the algorithm as part of XtreemFS. An extensive evaluation shows that the proposed algorithm has no severe impact on user I/O, and that it scales to large numbers of snapshots and versions.

Zusammenfassung

Viele moderne Dateisysteme unterstützen Snapshots zur Erzeugung konsistenter Online-Backups, zur Wiederherstellung verfälschter oder ungewollt geänderter Dateien, sowie zur Rückverfolgung von Änderungen an Dateien und Verzeichnissen. Während frühere Arbeiten zu Snapshots in Dateisystemen vorwiegend lokale Dateisysteme behandeln, haben moderne Trends wie Cloud- oder Cluster-Computing dazu geführt, dass die Datenhaltung in verteilten Speichersystemen an Bedeutung gewinnt. Solche Systeme umfassen häufig eine Vielzahl an Speicher-Servern, was besondere Herausforderungen mit Hinblick auf Skalierbarkeit, Verfügbarkeit und Ausfallsicherheit mit sich bringt.

Diese Arbeit beschreibt einen Snapshot-Algorithmus für großangelegte verteilte Dateisysteme und dessen Integration in XtreemFS, ein skalierbares objektbasiertes Dateisystem für Grid- und Cloud-Computing-Umgebungen. Die zwei Bausteine des Algorithmus sind ein System zur effizienten Erzeugung und Verwaltung von Dateiinhalts- und Metadaten-Versionen, sowie ein skalierbares, ausfallsicheres Verfahren zur Aggregation bestimmter Versionen in einem Snapshot. Um das Problem einer fehlenden globalen Zeit zu bewältigen, implementiert der Algorithmus ein weniger restriktives, auf Zeitstempeln lose synchronisierter Server-Uhren basierendes Konsistenzmodell für Snapshots. Dabei dehnt er den Zeitpunkt, zu dem alle Server ihren Zustand festhalten, zu einer kurzen Zeitspanne aus, deren Dauer von der Abweichung der Server-Uhren abhängt. Zudem wird sichergestellt, dass kausale Abhängigkeiten von Änderungen innerhalb von Snapshots erhalten bleiben.

Die wesentlichen Beiträge der Arbeit sind: 1) ein formales Modell von Snapshots und Snapshot-Konsistenz in verteilten Dateisystemen; 2) die Beschreibung effizienter Verfahren zur Verwaltung von Metadaten- und Dateiinhalts-Versionen in objektbasierten Dateisystemen; 3) die formale Darstellung eines skalierbaren, ausfallsicheren Snapshot-Algorithmus für großangelegte objektbasierte Dateisysteme; 4) eine detaillierte Beschreibung der Implementierung des Algorithmus in XtreemFS. Eine umfangreiche Auswertung belegt, dass der vorgestellte Algorithmus die Nutzerdatenrate kaum negativ beeinflusst, und dass er mit großen Zahlen an Snapshots und Versionen skaliert.

Contents

1 Introduction

File systems that are capable of taking snapshots have various advantages over conventional file systems. Their ability to capture and retain files and directories in their momentary states permits them to create consistent on-line backups as a means of recovering from data loss, to revert to prior states in the event of a corruption or inadvertent change of data, and to keep a record of changes and previous states for future analyses.

Throughout the last decades, the problem of taking snapshots in file systems has been extensively studied, and a multitude of snapshotting file systems have been presented. In recent years, however, the excessively growing volume of digital data has caused a similar growth in the scale of file system installations. Today, modern trends like cloud computing call for file and storage systems that accommodate the data of numerous different users, while file systems like the Google File System [GGL03] store data volumes at the scale of many petabytes on a single data center, so as to provide the core building block for Google's large-scale data management systems [CDG+06, BBC+11]. Moreover, most of the 500 largest supercomputers in the world resort to parallel file systems like Lustre [Clu02], Panasas ActiveScale [TGZ+04] or GPFS [SH02] to store and manage vast amounts of data.

As data volumes at such a scale exceed the capacity of single-host storage systems, distributed data storage has substantially gained in importance. Distributed file systems integrate storage, network and compute resources of multiple servers to provide for a higher storage capacity, I/O throughput and parallelism of accesses than traditional single-server file systems. Consequently, their architectures differ from the ones of traditional file systems. Conventional approaches, which involve that each data volume is entirely hosted by a single machine, have been replaced by concepts like SAN and object-based storage [FMN+05, MGR03]. With these architectures, data and metadata of a volume as well as the administrative control over these are shared between different host machines that may reside in different networks and data centers.

Since traditional snapshot techniques for file systems are confined to single machines, they are not applicable to modern distributed file systems. Instead, novel solutions are necessary that take the individual properties of distributed file systems into account.

1.1 Properties of Large-scale Distributed File Systems

File system installations for modern cluster and cloud computing environments provide huge amounts of storage capacity across numerous servers and storage devices. They differ from traditional small-scale and single-host file systems in various ways.

They tend to have:

1. **The ability to be scaled out.** Traditional small-scale file system installations are typically built upon a limited number of physical hardware components. The only way to increase capacity and performance is to *scale up* the underlying hardware i.e., to replace existing components with more powerful ones. As opposed to this, modern parallel and distributed file systems are capable of incorporating additional hardware resources i.e., being *scaled out*. Thus, incremental upgrades are possible at a low cost.

2. **Higher workloads and access rates.** Because of their capacity limitations, local and single-server file systems are limited in the number of users and requests they can handle in parallel. Large-scale distributed file system installations are less restrictive in this respect. They are typically accessed by many users and applications at a time, and accesses are generally more data-intensive.

3. **More restrictive availability requirements.** Owing to the fact that access rates are higher and more users are involved, the cost of downtimes is generally higher. Maintaining availability despite failures and downtimes of components is therefore essential.

4. **A higher susceptibility to failures.** Increasing the number of hardware components reduces the mean time between failures of individual components. In large-scale file system installations, component failures are the norm rather than the exception [GGL03, WBM+06]. Google's file system installations, which are backed by thousands of physical machines, permanently face outages and downtimes. Common reasons are defective hardware, software bugs and human errors [GGL03].

1.2 Implications on Snapshots

In consideration of these properties, a snapshot infrastructure has to fulfill various requirements in order to be suitable for large-scale distributed file systems. First, it needs to exhibit the same degree of *scalability* as the file system itself. In particular, it must be able to incorporate large numbers of file system hosts. Adding new hosts should not lead to an increased overhead in taking snapshots in terms of runtime, network traffic, disk I/O and computation, as this could limit the total scale of a file system installation and adversely affect the overall system performance.

As a consequence of the second and third property, it essential that the snapshot infrastructure be *non-disruptive*. While it is acceptable for most local file systems to briefly interrupt service in order to take a snapshot, such an interruption may cause extended downtimes in a distributed file system. Especially temporary unavailability of file system hosts as well as unpredictable network latency may lead to tedious and costly blocking periods, which are not acceptable.

The fourth property points out the importance of *crash resilience*. As a large number of individual components comes with a high failure rate, it is important to reduce the impact of host failures on snapshots to an unavoidable minimum. A distributed snapshot infrastructure should remain operable in the face of failures. Snapshots of data that are unaffected by failures should remain accessible, and unreachable components should not inhibit new snapshots.

1.3 Challenges

The term "*snapshot*" originally comes from the domain of photography, where it has the meaning of a spontaneously taken picture of a scene. Because of the properties of such a picture i.e., the fact that it captures many different items and incidents at a particular instant, the term has been applied to many different fields in computer science. In the context of databases, file systems, distributed systems and applications, a snapshot is typically used as a synonym for an image of the global system state at a certain time.

In the context of a file system, a snapshot corresponds to a collection of files and directories in their momentary states at a particular point in time. Recording snapshots of a distributed file system bears two core challenges, which can be described by the following questions:

1. How can previous states of the file system be *preserved* in the face of ongoing changes?
2. How can the file system ensure that different parts of its state are captured *simultaneously*?

We refer to these as the challenges of *version management* and *snapshot consistency*.

1.3.1 Version Management

A snapshot has to reflect an immutable image of the file system. Regardless of any concurrent or future changes, accessing files and directories in a snapshot has to deliver the same results. This property is occasionally referred to as *snapshot isolation* [BBG+95] in the context of database transactions.

To ensure immutability, data needs to be protected from being overwritten or deleted. Prior versions of files and directories have to be retained, which remain unchanged in the face of changes to the original data. This calls for version management mechanisms, which decide when to create versions, how and where to store them, and how long to retain them. Accordingly, it is necessary to compare and evaluate version management schemes with respect to their potential uses in a distributed file system.

1.3.2 Snapshot Consistency

Snapshots are generally expected to reflect an image of all files and directories at the same instant everywhere in the system. On a local file system, this can be accomplished

fairly easily. If all state resides on a single machine that performs all changes in their chronological order, it is easy to determine if a change took place before or after taking a snapshot. On a distributed file system that spreads its system state across a potentially large number of hosts, the problem becomes complex. Such systems are asynchronous and lack a global time or centralized sequencer that enforces a global total order on state changes. This effectively makes it impossible to detect if states on different hosts were changed at the exact same point in time.

To overcome this immanent limitation, it is necessary to find a meaningful alternative to the restrictive requirement that states be captured at the very same instant. From an external observer's point of view, it is sufficient to guarantee that local states captured in a snapshot *could have* existed at the same point in time in the past. A global state with this property is generally referred to as a *consistent* state.

As a consequence of these considerations, it is necessary to define a consistency model that is in line with the notion of simultaneousness from an external observer's point of view. Such a consistency model must be enforceable in respect of the properties and requirements of a distributed file system in terms of scalability, non-disruptiveness and fault tolerance. Snapshot algorithms for distributed systems, as e.g. proposed by Chandy and Lamport [CL85], Lai and Yang [LY87] or Mattern [Mat93], enforce a causality-based consistency model that involves a reasonable abstraction of a global time. However, their scalability and failure tolerance is generally limited, which makes it necessary to investigate novel approaches.

1.4 Goal and Approach

The goal of this thesis is to present a comprehensive solution to the problem of capturing snapshots of large-scale distributed file systems. We follow the approach of bringing together snapshotting techniques of local file systems with distributed snapshot algorithms. We subdivide the problem of taking snapshots into the aforementioned problems of version management and snapshot consistency, for which we analyze and investigate eligible techniques and algorithms. The result is a distributed infrastructure for file system snapshots that incorporates the distinctive properties of large-scale distributed file systems.

1.5 Contributions

The thesis makes various contributions, the most important ones being:

- the description of a scheme for the management of versions in object-based file systems, which covers metadata and file content. In particular, we introduce and describe BabuDB [SKHH10], a database system developed for the management of file system metadata. BabuDB maintains metadata of files and directories in a particularly fast and efficient manner and allows to capture local metadata snapshots instantaneously. Besides, we describe how version management techniques

like *copy-on-write* and *logging* can be used to maintain versions of files and objects in an object-based [FMN+05, MGR03] file system.

- the formal specification, analysis and comparison of consistency models for snapshots in distributed file systems. Based on a formal model of a distributed file system, we introduce and compare different consistency models. We present the formal description of a consistency model that is based on loosely synchronized server clocks. The model can be implemented in a scalable and failure-tolerant fashion and allows to capture a snapshot within a constant, short period of time, regardless of the number of servers and their availability.

- a detailed, formal description of a scalable, failure-tolerant and non-disruptive snapshot algorithm for large-scale object-based file systems. The algorithm superimposes a versioning scheme on the local data management subsystem of each file system server. Individual versions of file content and metadata residing on different hosts are bound snapshots by means of timestamps, which originate from loosely synchronized, local server clocks. Snapshots can thus be captured within a narrow time frame that depends upon the maximum clock drift between all servers.

- the practical implementation and evaluation of the algorithm. We developed the algorithm as part of XtreemFS [HCK+07], a scalable and failure-tolerant object-based file system. We address practical aspects and challenges attached to the implementation, which involve the retention of versions, the processing of atomic data modifications across multiple servers, as well as the interplay of snapshots and replication.

1.6 Outline

The rest of this thesis is structured as follows:

Chapter 2 provides the background for the work presented in this thesis. It starts with an analysis of the basic functional and architectural properties of file systems, discusses architectures for distributed file systems with a particular focus on object-based storage, and introduces the properties and architecture of XtreemFS.

Chapter 3 addresses the problem of version management in file systems. It gives an overview of schemes and techniques for the management of file content and metadata versions and presents versioning schemes for file content and metadata in XtreemFS. It concludes with a comprehensive study of previous work on file systems with versioning and snapshotting capabilities.

Chapter 4 addresses the challenges of snapshot consistency. It introduces the consistency models of *point-in-time consistency*, *causal consistency* and *consistency based on loosely synchronized clocks*. It describes these on the basis of a formal model

of a distributed file system, and analyzes and compares their individual properties. Particular emphasis is placed on the constraints of the consistency models in terms of availability, scalability and fault tolerance, which define their suitability for large-scale distributed file systems.

Chapter 5 provides a formal description of a scalable and failure-tolerant algorithm for snapshots in object-based file systems. Based on the results from the previous chapters, it presents a formal model of an object-based file system along with a formal description of the necessary extensions for version management and consistency.

Chapter 6 addresses practical challenges regarding the implementation of snapshots in XtreemFS. It gives an overview of the implementation and presents solutions to the problems of redundant version cleanup, atomic modifications and the integration of snapshots with replication.

Chapter 7 presents an extensive experimental evaluation of our implementation. On the basis of artificial workloads and real-world traces, it analyzes different performance and scalability characteristics of the data, metadata and version management infrastructures in XtreemFS. The results confirm the viability of our approach.

Chapter 8 presents a summary of results and future work perspectives.

2 Large-Scale Distributed File Systems

The rapid growth of data volumes over the last decades has leveraged diverse architectures for distributed data management systems. Aside from traditional schemes like NAS and SAN, the concept of *object-based storage* [FMN+05, MGR03] has caused a paradigm shift in the design of distributed file systems. Object-based file systems are composed of independent, intelligent, loosely-coupled storage servers, which set the scene for scalable, cheap and extensible file system installations and thus provide a solid basis for algorithms and techniques to capture snapshots in large-scale distributed file systems.

This chapter contains background information for the thesis. After introducing the basics of file systems (2.1), it presents a comprehensive outline of distributed file system architectures (2.2). Particular focus is placed on the concept of *object-based storage* (2.3) and the architecture of XtreemFS (2.4), which provide the basis for the snapshot infrastructure presented in chapter 5.

2.1 Concept and Design of File Systems

File systems are present on nearly any of today's computing devices. They are the basic repository for the data of operating systems, applications, users, and alternative data management systems like databases. The main reason for the success of file systems is the fact that they are based on simple, intelligible concepts. Files and directories provide a lean yet powerful abstraction that corresponds with the natural way of categorizing, naming and memorizing things. Descriptive path names identify chunks of data in an unambiguous manner, which makes it easy to retrieve and access them. Directories make it possible to organize related files in logical groups, which in turn can be arranged in a hierarchy to simplify their retrieval.

As a result, concepts and interfaces of file systems have barely changed since the early years of electronic data processing. Application developers and users alike are generally acquainted with the interface and concepts of file systems, which makes file systems the most convenient way of storing data in a structured manner.

2.1.1 POSIX Interface and Semantics

To ensure interoperability between applications and file systems, different standards have emerged that describe the interface and semantics of file system operations. A prominent standard is the *Portable Operating System Interface for Unix* (POSIX) [IEE08]. POSIX specifies the Unix operating system API, which involves a description of the

`int open( const char *path, int oflag, ... )`	Opens a file located at the given `*path`. If successful, a file descriptor is returned, by means of which the open file can be identified in the context of other operations. `oflag` specifies a set of open flags that define the open mode, such as "read-only", "write-only", or "read-write".
`int read(int fildes, void *buf, int nbyte)`	Reads up to `nbyte` bytes from an open file into a buffer `*buf`, starting at the current marker position. After successfully reading a file, the marker is moved behind the last byte read.
`int write(int fildes, void *buf, int nbyte)`	Writes up to `nbyte` bytes from a buffer `*buf` to an open file, starting at the current marker position. After successfully writing a file, the marker is moved behind the last byte written.
`int close(int fildes)`	Closes a file. This invalidates the file descriptor and frees all internally attached resources.
`int truncate( const char *path, off_t length)`	Sets the length of the file located at `*path` to `length`. Any data beyond offset `length` is discarded. If `length` is beyond the end of the file, any subsequent access to the extended range of bytes behaves as if they were zero-filled.
`int stat( const char *path, struct stat *buf)`	Retrieves information about the file located at `*path` and stores it in `*buf`. The resulting buffer contains data like owner and owing group ID, timestamps, file size and access mode.

Table 2.1: Selection of important POSIX file system operations

names, parameters and semantics of all operations in the file system interface. Some of the most important file system operations defined in the POSIX standard are listed in table 2.1.

2.1.2 File System Design

Although there are substantial differences in the internal design of different file systems, they all have in common that they offer the abstraction of files and directories as a uniform access scheme. Internally, they resort to underlying storage devices to persistently store data. Storage devices are typically *block devices*; they expose a low-level interface that provides read and write access to a contiguous range of physical *blocks*. We refer to such an interface as a *block-based* interface. Blocks have a fixed size and constitute the smallest units at which data can be written to or retrieved from the device. Often, blocks relate to physical regions on the device. All blocks on a device are contin-

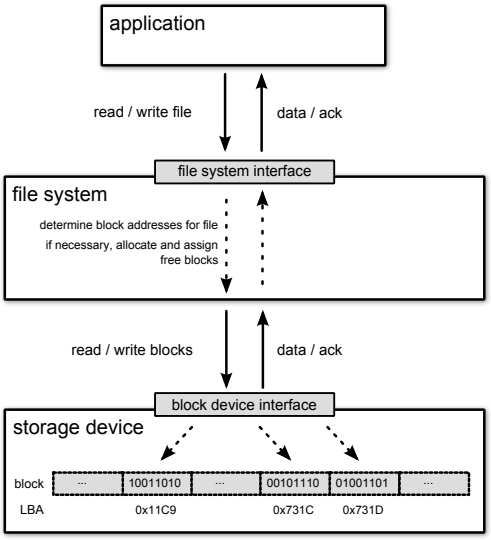

Figure 2.1: Interactions between applications, file systems and storage devices

uously numbered with *logical block addresses* (LBAs), which make it possible to address each block individually. Practically all of today's storage devices that are based on tape, flash memory or rotating disks offer a block-based interface, such as ATA or SCSI.

File systems are responsible for the management of blocks on the underlying storage device. Since block-based storage devices do not impose any structure on the data that resides on them, the file system has to keep track of free and occupied blocks and the assignment of blocks to files. Figure 2.1 shows an example that illustrates the principle. In response to a file access triggered by an application, e.g. a POSIX `read` or `write` call, the file system determines the LBAs of the blocks containing the file content and reads or writes the data at the storage device. In case of a `write` request, it may have to allocate new blocks from the pool of free blocks to the file, in order to store the data to be written. These steps are invisible to users and applications, as the file system interface only offers high-level operations like reading and writing byte ranges of files. In essence, the file system translates high-level access requests to low-level interactions with a storage device and abstracts from individual properties of the storage hardware.

2.1.3 File Content and Metadata Management

We refer to the data stored in a file as the file's *content*. File systems render file content accessible as a contiguous range of bytes, where each byte is identified and addressed through a logical *offset*, a number that defines the position of the byte relative to the first byte of the file. File systems internally map these logical offsets to offsets of individual blocks on the storage device.

Besides the content of files, file systems need to manage auxiliary data in order to maintain information about files and directories. We refer to this as *metadata*, a collective

term for any data stored in the file system except file content. Metadata covers the physical locations (such as block addresses) of file content, the directory tree with the names of all files, security-related information about files and directories like access rights and ownership, extended attributes, access timestamps, and various other pieces of information.

The file system has to be able to quickly store and retrieve the metadata and content of a given file. For this purpose, many modern file systems use index structures, which are often stored at predefined block locations on the underlying storage device. On Unix systems, the metadata of each file is stored in a dedicated fixed-size metadata record called *index node* (*inode*). Apart from user-accessible metadata, each inode stores the addresses of the disk blocks containing the file content. The size of an inode is limited, usually to 128 or 256 bytes. Thus, large files may require the use of indirect blocks in order to reference the content of a file, which store block addresses for file content instead of the file content itself. Very large files may use multiple such levels of indirection.

2.2 Distributed File System Architectures

Over the last decades, the volume of digital data has increased dramatically. The scale of storage systems for commercial and scientific applications has grown by many orders of magnitude [BHS09]. As a consequence, the distributed storage of data has substantially gained in importance, and a multitude of distributed file systems have been developed that aggregate storage resources across different sites and storage devices.

A substantial feature of distributed file systems is distribution transparency. By some means or another, all distributed file systems aim to relieve users and applications of the task of dealing with distribution aspects. This includes the allocation and physical deployment of data and metadata across the different nodes, as well as the communication required to access data on remote nodes. Distributed file systems typically offer a high-level POSIX-like interface and hide all internals from their users. A *client* component renders the file system accessible on a node, generally in the form of a virtual storage device or mount point.

The need for scalable storage has leveraged different design patterns for distributed and parallel file systems, which can be roughly categorized into *decentralized approaches*, *SAN*, *NAS* and *object-based storage*.

2.2.1 Decentralized File Systems

A common approach to build distributed systems at a large scale is to connect numerous equivalent components in a decentralized, loosely coupled organization scheme. Especially academia has made this approach popular with fields of research like peer-to-peer and distributed hash tables. Decentralized architectures are typically considered as scalable and resilient alternatives to centralized ones.

The need to provide storage capacity at a large scale has fostered the development

of decentralized file systems. As a result, globally distributed peer-to-peer file systems like Farsite [ABC+02], OceanStore [KBC+00], Pangaea [SKKM02] and Ivy [MMGC02] have been developed, which share the goal of aggregating user-provided storage resources to a large virtual pool. A typical assumption of such decentralized file systems is that storage resources are untrustworthy, unreliable and not permanently connected. The core problems related to decentralized file systems are hence availability, data safety and security, which are generally addressed by means of replication as well as encryption and authentication protocols.

In view of the fact that solutions to these core problems increase the overhead associated with data accesses, peer-to-peer file systems typically exhibit a lower performance than their centralized counterparts. Requests to read or write a file may have to be routed over multiple hops, which may substantially increase latency and consume additional network bandwidth. For this reason, decentralized file systems have failed to reach a wide acceptance.

2.2.2 Storage Area Networks

A widespread storage solution for data centers are *storage area networks* (SANs). The SAN concept was introduced to aggregate many individual network-connected block storage devices to a large virtual device, which allows data access in a highly parallel fashion.

Block storage devices and clients in a SAN are typically interconnected through a dedicated high performance local area network that relies on fiber channel or fast Ethernet connections. To guarantee a high-throughput and low-latency data access, the network is exclusively used for storage-related data transfers. Protocols like *iSCSI* [SMS+04] or *SCSI over Fiber Channel* make the native block interfaces of all individual storage devices accessible through the network. SANs can be scaled out by adding new storage devices, which increases the total capacity as well as the maximum read and write throughput.

Most file systems that run on top of a SAN support block-level striping to enable a high-throughput parallel access to the data stored across the remote storage devices. Besides performing block allocations and metadata management in a distributed fashion, a SAN file system has to ensure consistency despite concurrent modifications of file content and metadata. Possible solutions are central components that coordinate conflicting changes, or distributed locking mechanisms [SH02]. Accordingly, the tasks of mapping files to blocks as well as allocating free blocks to new files are either performed by a central management node, or in a distributed, coordinated manner among the nodes themselves. Prominent examples of SAN file systems are GPFS [SH02], Frangipani [TML97], Oracle Cluster File System 2[1] and GFS [SRO96].

While SAN enables fast data access, an important limiting factor of the SAN architecture is lack of security [ADF+03]. SAN storage servers cannot authenticate and authorize requests, as their functionality is effectively restricted to block-level read and

[1] `http://oss.oracle.com/projects/ocfs2/`

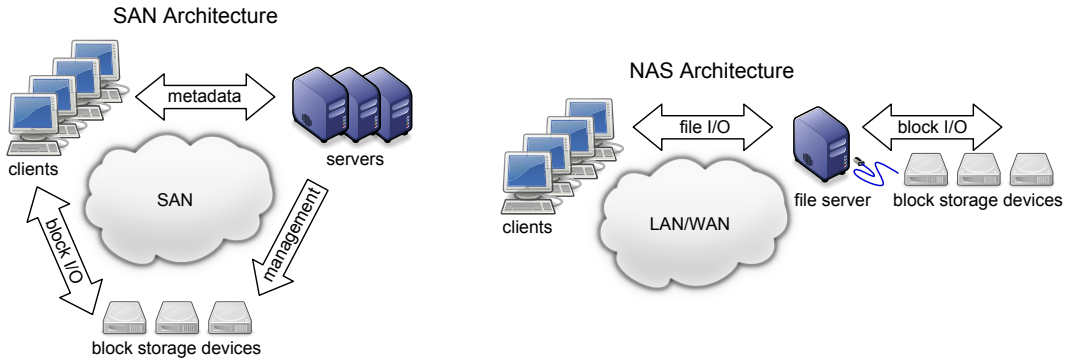

Figure 2.2: Comparison of SAN and NAS architectures

write access. Although mechanisms like *zoning* and *LUN masking* make it possible to restrict client access to a subset of data volumes and thus to enforce a coarse-grained security model, file level security depends on the trustworthiness of all clients and the underlying network. These missing security-related features, along with the fact that the spatial extent of the underlying high-performance network is physically limited, generally restrict SAN installations to single data centers.

2.2.3 Network-Attached Storage

A design pattern that overcomes the limitations of the SAN concept in terms of security and spatial extent is *network-attached storage* (NAS). The primary motivation of NAS is to enable a secure and uniform remote access to data in a heterogeneous environment.

Figure 2.2 shows a comparison of NAS and SAN architectures. As opposed to SAN, NAS offers remote data access on the level of files and directories rather than blocks. Storage servers are responsible for their individual sets of volumes, where each volume represents an independent file system along with file content and metadata. A volume is typically backed by a local file system on its server but may also be backed by a SAN file system running on an underlying SAN. A storage server exports its volumes to the network through high-level device-independent interfaces, such as NFS [SEN10], CIFS/SMB[2] or HTTP [FGM+99]. Common examples of NAS file systems are NFS implementations, AFS [HKM+88], Coda [Bra98], Samba[3] and FreeNAS[4].

NAS solutions usually rely on network interconnects that are based on commodity hardware, such as Ethernet and TCP/IP. For this reason, they are generally cheaper and easier to maintain than SAN solutions with purpose-built hardware, but also provide a lower performance. The performance penalty also comes from the fact that the network is not exclusively dedicated to storage but may be shared for different purposes. Access performance to individual volumes can only be increased by scaling up the existing hardware, as each volume resides on its own server.

[2] http://tools.ietf.org/html/draft-leach-cifs-v1-spec-01
[3] http://www.samba.org/
[4] http://freenas.org/

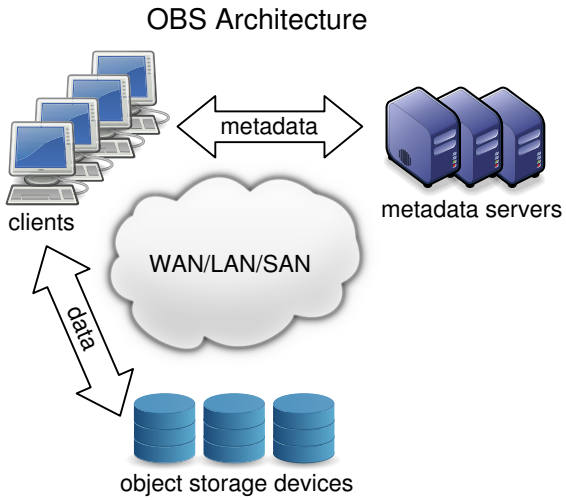

Figure 2.3: Object-based storage architecture

2.3 Object-Based Storage

In recent years, distributed and parallel file system design has undergone a paradigm shift. With more compute power becoming available on storage devices, *object-based storage* [FMN+05, MGR03] has emerged as a flexible and scalable alternative to traditional architectures for distributed file systems. The concept describes a convergence of NAS and SAN: as with NAS, files are read and written over the network through a high-level interface in a secure and platform independent manner; as with SAN, clients communicate with storage servers directly rather than through a head node, thus offering SAN-like scalability and performance characteristics. An overview of the architecture is given in figure 2.3.

2.3.1 Architecture

As the name suggests, the concept of object-based storage relies on the abstraction of an *object* as a means of storing, accessing and transferring data. An object is a container that encapsulates a range of bytes associated with a file. Unlike blocks managed by a block storage device, however, objects are flexibly-sized units that are independent of the underlying storage device. The interface through which objects are accessed and modified resembles the one of a file system rather than a block device, as objects are attached to device-independent identifiers and can be read and written at byte granularity. The difference between the traditional block-based storage model and the object-based storage model is illustrated in figure 2.4.

Object Management. Objects are managed by *object storage devices* (OSDs). OSDs provide an interface to store and access objects over a network. Unlike network-attached block storage devices from the SAN architecture, however, they possess the capacity to

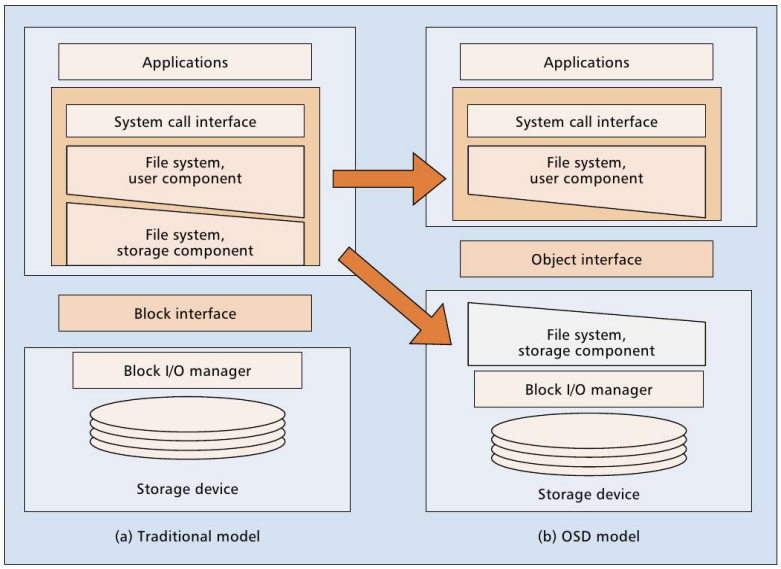

Figure 2.4: Comparison of the traditional block-based storage model (left) and the object-based storage model (right), taken from [MGR03].

perform computations, which empowers them to accomplish more sophisticated tasks than simple I/O of data blocks. Low-level storage operations like allocating, storing and retrieving blocks on the underlying storage device are internally performed by the OSD.

Besides reading and writing objects, the range of functions offered by an OSD may cover virtually any aspect of data management. Examples are enhanced capabilities like checksumming, encryption, or filtering of data; functionality to ensure data safety like replication and RAID; versioning and snapshotting functionality; as well as maintenance support like data scrubbing.

Metadata Management. While object-based file systems share the same concept of objects and OSDs, they differ in their management of metadata. One solution to the problem of metadata management is to attach metadata to objects in the form of attributes, which are stored together with the file content across the OSDs. Such a design is suggested in the T10 [NCI04] standard for object-based storage systems and has been implemented in the Panasas ActiveScale file system [TGZ+04]. It involves a natural distribution of metadata, which helps to balance load across all servers.

However, the distributed management of metadata comes at the cost of an increased complexity of metadata-related operations. Especially the implementation of operations that affect directories offers significant challenges, as they require atomic operations across multiple OSDs [ADD+08a]. A common example is the creation of a file that has to be linked to a directory residing on a different OSD. This makes it necessary to use costly and complex locking mechanisms.

A common way to circumvent such problems are dedicated *metadata servers* (MDSs),

which are exclusively responsible for the management of metadata. The range of functions provided by an MDS depends on its design and architecture [ADD+08b] but typically covers the maintenance of the directory tree together with the mapping of files to directories, which involves the creation, deletion and retrieval of information about files as well as the enforcement of access control on the file system.

Access. *Clients* provide users and applications access to their data through a common file system interface. They receive file system-related calls from their local operating systems, such as `open`, `read` or `write`, and translate them into interactions with the respective MDSs and OSDs. For instance, when a user or application executes an `open` call, the client first contacts the MDS in order to resolve the file's path name to the network addresses of the OSDs that are responsible for the file's content. Any subsequent `read` or `write` calls can then be performed directly on these OSDs. If files are composed of multiple objects, the client may further need to transform a `read` or `write` request into multiple individual object-related requests. Clients often use local caches in order to reduce the latency and network traffic caused by interactions with the servers.

Security. Object-based file systems typically do not assume that clients or networks are trusted. Accordingly, the authentication and authorization of users needs to be enforced on the server side. MDSs thus perform authorization checks before executing metadata-related operations on behalf of a user. This implies that users are authenticated in a secure manner, which, e.g., can be ensured by means of X.509 [CSF+08] certificates. In accordance with the POSIX standard, operations that involve interactions with OSDs, such as reading or writing files, are authorized by the MDS when the file is opened. Upon a successful authorization, the MDS responds with a *capability* [Gob99], a cryptographically secure, signed access token, which is sent to the OSD with each subsequent change to the file content. Aside from ensuring that the capability provides sufficient access rights to perform the requested operation, the OSD verifies the capability signature, which, e.g., can be done through a shared secret between OSDs and MDSs. Capabilities generally have a limited validity time span in order to make sure that access rights can be revoked.

2.3.2 Advantages and Relevance

The concept of object-based storage opens up new prospects for distributed file systems. By delegating responsibility for the block management from a central file server to a distributed set of storage nodes, load is shifted away from the critical access path and distributed among the nodes. Also advanced functionality that goes beyond providing read and write access can be performed in a decentralized manner, which makes it easier to scale out an object-based file system installation. This applies to features relating to objects, like parallel I/O and striping of files (which requires objects to be distributed across multiple OSDs), replication and versioning of file content, quality of

service, data integrity checks, pre-allocation of storage space, and many more. Server-enforced security mechanisms allow for a deployment in an untrusted environment, which enables globally distributed organization-spanning installations.

Because of its flexibility and scalability, object-based storage has become one of the prevalent design patterns for modern parallel and distributed file systems. The majority of the top 500[5] supercomputers in the world use object-based file systems like Lustre [Clu02] or Panasas Active Scale [TGZ+04] to accommodate their demand for scalable storage.

The topic of object-based storage has also been addressed by the research community, which has brought out a range of object-based file systems including NASD [GNA+98], Antara [ADF+03], PVFS [CIRT00], Ceph [WBM+06] and XtreemFS [HCK+07]. Issues and challenges related to object-based storage have been extensively discussed in literature, including striping and parallel I/O [TGZ+04, SKH+08], metadata management and distribution [BMLX03, WPBM04a, WPBM04b, ADD+08b, ADD+08a], replication and data placement [HM03, HM04, WBMM06], data management in the OSD [Wan06, WBML04, Wei04], security and authentication [Gob99, OM05, LMJ07], access control [PB05b], management of extended attributes [DDWA07], quality of service [WB07], and the enforcement of quotas [PLG+07].

While object-based storage has increasingly gained in importance, efforts have been made to standardize concepts and interfaces of object-based storage systems. As an extension of the SCSI standard for storage devices, the ANSI T10 standard [NCI04] aims to formally specify the architecture, interface, protocols and functionality of an object-based storage system. Also parallel NFS (pNFS), an advancement of the NFS standard that supports parallel I/O, takes object-based storage into account [HWZ10].

2.4 XtreemFS

XtreemFS [HCK+07] is a distributed file system for grid and cloud computing workloads. It has been designed to safely store large data volumes across heterogeneous, inexpensive off-the-shelf servers. which are connected via an arbitrary wide-area network such as the Internet. Characteristic properties of XtreemFS are scalability, crash resilience, high availability, security and extensibility. XtreemFS has been developed since 2006 and is available as a multi-platform open-source software[6].

2.4.1 Architecture and Components

In accordance with the concept of object-based storage, XtreemFS has a modular client-server architecture, which is shown in figure 2.5. Clients and servers are connected via an IP-based network with no specific requirements in terms of security, failure tolerance and performance. Servers can be run on arbitrary commodity hardware with the ability to perform computations, to persistently store data, and to communicate with remote

[5] http://www.top500.org
[6] http://www.xtreemfs.org

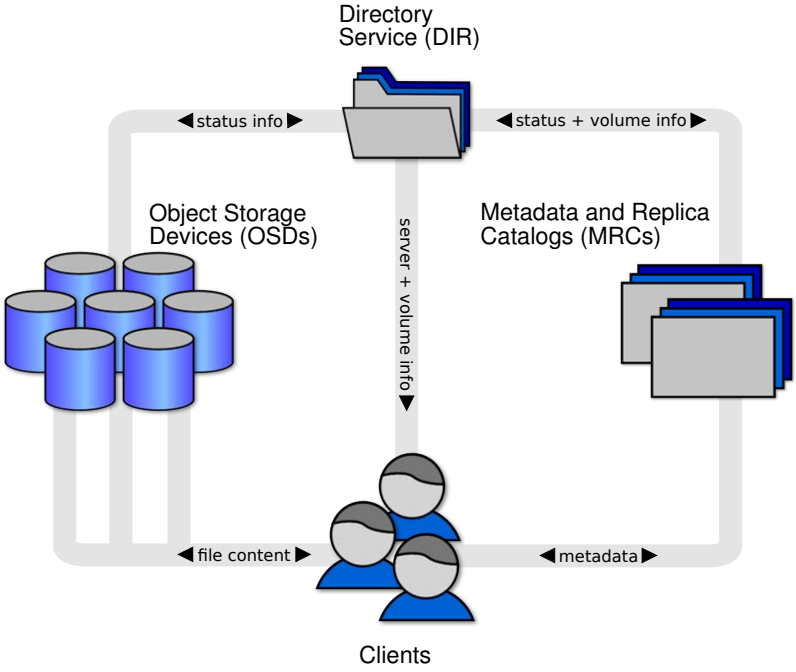

Figure 2.5: XtreemFS architecture

clients and servers. Files consist of objects and metadata, which are managed by storage and metadata servers. A separate client component renders XtreemFS accessible through a POSIX file system interface.

OSD. Object Storage Devices (OSDs) store their objects as files on a locally mounted file system. Objects pertaining to the same XtreemFS file are sequentially numbered and reside in the same directory on the local file system, which is named after the file's globally unique ID. This allows for a quick and easy retrieval of individual files and their objects. By spreading objects of a file across multiple OSDs, XtreemFS supports striping [SKH+08].

MRC. Metadata and Replica Catalogs (MRCs) are responsible for the management of metadata and the maintenance of replica sets in the file system. The file system metadata is subdivided into individually mountable volumes with individual settings and policies that define how features like access control, replication, and the assignment of OSDs to files are implemented.

The data storage back-end in an MRC is based on BabuDB [SKHH10], an optimized, high-performance non-relational database system. Each volume is stored in a separate internal database, and the hierarchical directory tree with the metadata of all nested files is mapped onto a flat namespace consisting of key-value pairs. The metadata of each file contains a set of file replicas, which in turn consist of a list of OSDs and a striping pattern that describes how the file content is dispersed over the OSDs. Such a

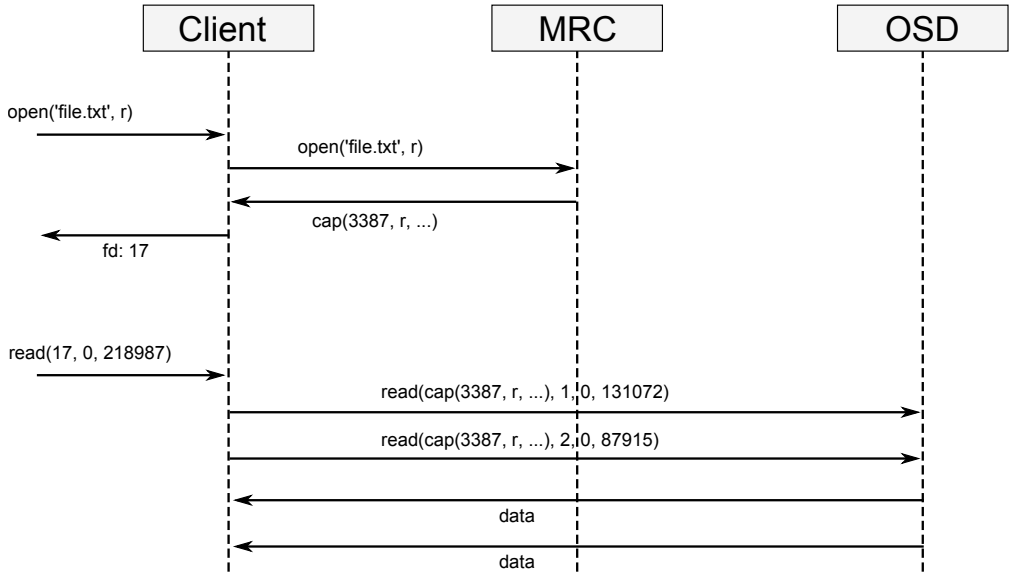

Figure 2.6: Interaction between the XtreemFS client and servers when opening and reading a file

replica set allows for an individual assignment of OSDs to replicas and replicas to files.

DIR. The Directory Service (DIR) provides a central registry for XtreemFS services and volumes. Service instances register at the DIR and report their status in regular intervals, so as to allow clients and other services to look up relevant information, such as liveness, remaining storage space and current load. The DIR holds a list of all existing volumes throughout the installation. When mounting a volume, a client first contacts the DIR to retrieve the communication endpoint of the MRC responsible for the volume's metadata.

Client. Clients make XtreemFS volumes accessible to users and applications through mount points, which behave like local file systems. They translate system calls on a mount point into an interactions with the servers. Figure 2.6 illustrates the sequence of interactions initiated by the client when an application opens and reads a file, which involves an `open` call on the MRC, followed by a sequence of `read` calls on the respective OSDs. Other capabilities of the client involve automatic fail-over when servers are unavailable, as well as caching to reduce latency when repeatedly accessing files and directories.

2.4.2 Target Environments and Characteristics

The initial goal of XtreemFS was the development of a file system for grid computing environments [HCK+07, HCK+08]. Common grid data management systems at that

time had significant shortcomings in terms of efficiency, performance and usability. We used this insight as an opportunity to come up with a novel, better solution to the problem of maintaining large amounts of data in a compute Grid. With the advent of cloud computing [AFG+09], the focus and target environment of XtreemFS was shifted from grids to clouds. As a consequence, we had to design and develop XtreemFS with different core properties in mind, the most important ones being:

- **Scalability.** An XtreemFS installation can be scaled out by adding new servers. This increases storage capacity, attainable throughput and the number of users and requests that can be served concurrently. Scaling out an installation requires little administrative work, as new servers are integrated automatically once having been set up.
- **Reliability.** XtreemFS all keeps data safe and remains accessible in the face of downtimes and failures of individual system components. Typical failure scenarios in a distributed system like server crashes or network splits are transparently handled by the file system, thus conveying the impression that they never happen. To guarantee reliability, XtreemFS makes use of replication techniques.
- **Extensibility.** The object-based design of XtreemFS eases the integration of new features. In particular, the ability to store data and perform computations across all servers ensures that additional functionality of any kind can be added. The scale-out principle paves the way for doing this in a decentralized and scalable fashion.

As these core properties match with the requirements identified in section 1.2, we argue that XtreemFS provides a solid basis for the implementation and evaluation of snapshots in large-scale distributed file systems.

3 Version Management in File Systems

A fundamental capability of all snapshotting file systems is the retention and management versions. As opposed to a snapshot, which reflects a certain state of *all* files and directories, a version reflects a single piece of this state, such as an individual file or storage block. While snapshots are typically recorded in response to user requests or specific periodically recurring events, versions are recorded when data is modified. Once created, versions remain immutable, which implies that subsequent modifications cause new versions to be created. Accordingly, the set of versions of a data item reflects its history of changes.

Aside from being necessary for snapshots, versioning involves various advantages for data management systems. In general, the benefits can be grouped into three different categories i.e., recovery from user mistakes, recovery from system corruption, and analysis of historical changes [SGSG03, FB04]. Versioning techniques thus find a use in a large number of data management and file systems.

Previous work on file system snapshots and versioning has revealed that there are a wide range of different approaches and techniques for the management of versions in file systems. In particular, they differ in their answers to the questions *how*, *where* and *when* new versions are created and maintained. To preserve prior states in the face of arbitrary changes, they have to cover file content as well as metadata.

The chapter provides a summary of approaches to the management of versions in file systems (3.1) and gives a description of specific solutions to the problem of metadata versioning (3.2). It describes the internal design of the storage components in XtreemFS, where it addresses the management and versioning of file content (3.3) and metadata (3.4). It concludes with a summary of related work in the area of snapshots and version management in file systems (3.5).

3.1 Maintaining Versions in a File System

We identified three core questions, which help to specify the properties of a file system versioning infrastructure. These are:

- *How* are versions created, accessed and physically represented?
- *Where* do these versions reside?
- *When* is it necessary to create and delete versions?

In the following, we will analyze and discuss answers to these questions, which are known from previous work in the area of file system snapshots and versioning.

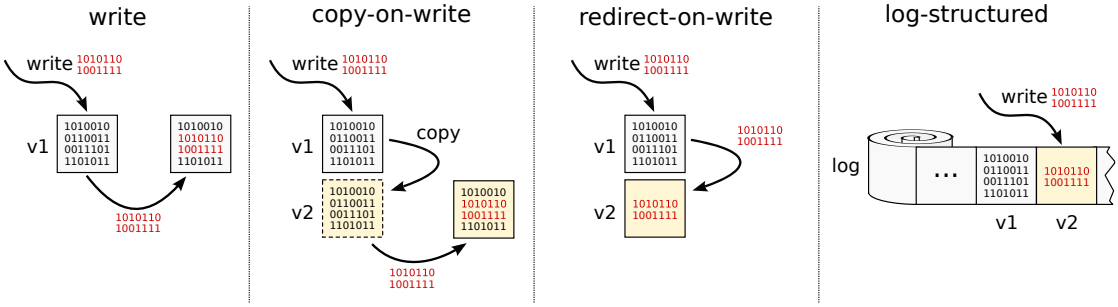

Figure 3.1: Comparison of versioning techniques. While the common implementation of a `write` operation (leftmost diagram) simply overwrites existing data, versioning-aware implementations preserve existing data by creating new versions.

3.1.1 Versioning Techniques

The management and retention of versions in file systems requires specific techniques to protect previous states from being overwritten. We refer to such techniques as *versioning techniques*. Various such techniques are present in modern file systems. Figure 3.1 illustrates different versioning techniques known from previous work.

Copy-on-write. The idea of *copy-on-write* (COW) is to create a copy of the latest version before applying changes to it. The copy retains the original data, whereas the new version contains the modified data. COW is one of the most widespread versioning techniques and has been implemented in a range of storage systems [HKM+88, CAK+92, HLM94, SFHV99, PB05a, SH02, BM07, MAC+08].

As COW may cause considerable chunks of data to be duplicated, it often comes with optimizations to diminish overhead and additional storage consumption. An optimization proposed by *Muniswamy-Reddy et al.* [MRWHZ04] is *copy-on-change*, which only creates physical copies of previous versions if data was actually modified. While *Muniswamy-Reddy et al.* compare the data to be written with the data to be replaced in order to determine if a copy needs to be created, *Quinlan and Dorward* [QD02] suggest to detect disparities between versions by means of hashing. When a new version is created, a hash value of its content is calculated by means of a collision-resistent hash function, which is used to address the version. Thus, a new version is only created if its hash differs from any other hash in the system. The concept of addressing and coalescing versions via hashes has become popular under the names *content-addressable storage* and *data deduplication*. It is first and foremost present in archival storage systems [UAA+10] where data is generally written only once, as hash calculations prior to every write operation may lead to a slightly increased latency of write accesses.

Redirect-on-write. While COW creates a physical copy of the current version and partially overwrites it, *redirect-on-write* (ROW) creates a new version that exclusively con-

tains the change. This eliminates the additional overhead of creating copies and thus allows for a better write performance. The original version remains unchanged, whereas the current version is composed of the original version and the change. ROW implicates that reading the current version also requires former versions to be read. This adversely affects read performance especially if consecutive versions are fragmented across the storage device. A possible way of mitigating the problem is to combine ROW with caching mechanisms, such that latest versions are also accessible through a cache. An architecture for a network file system following this approach has been presented by *Chadha and Figueiredo* [CF07].

Since the reading overhead is increased with each new version, the life time of a ROW-based snapshot is often also limited to the time it takes to create a backup, and versions are eventually coalesced again. *Xiao et al.* [XLY+06, XYR+09] showed in two comparative performance studies that ROW is generally more suitable for write-intensive workloads, whereas COW is for read-intensive workloads.

Log-Structured Approaches. The principle of recording changes without creating copies of previous state is also immanent in *log-structured* data management schemes. Log-structured file systems store all files and directories including data and metadata as a sequence of changes, which are recorded in a system-wide append-only log. Each change causes a corresponding log entry to be appended to the end of the log. This approach can be seen as a special case of ROW, where only a single log exists for all changes, which is typically stored in a contiguous region on the storage device. Thus, modifications can be applied to arbitrary files and directories through append writes at device level, which, compared to random block writes, are generally fast on rotating disks.

Since old log entries remain unchanged when data is written or deleted, the log contains the entire history of changes. Each entry along with all previous entries represents a distinct snapshot of the entire system. To avoid the traversal of all earlier entries in order to access a specific version of a file or directory, log-structured file systems typically optimize read access to data. *Rosenblum and Ousterhout* [RO91] suggest to maintain additional data structures that map inodes and data blocks to those log entries containing the latest versions. As such data structures can be cached, they are accessible fast in the general case. Similarly, *Whitaker et al.* [WBW96] suggest to use an additional B-tree-like data structure in order to map the virtual file address space to the log.

Aside from log-structured file systems, logging techniques are used in a range of other versioning file systems as a means of retaining prior versions [SGSG03, BCD04, MRWHZ04, AKM+07]. Most such file systems use a per-file log to record changes in their chronological order. If the latest version of a file or directory is accessed more frequently than prior versions, *Cornell et al.* [BCD04] suggest to use an *undo log* instead of a classical redo log. The undo log records inverted changes, which contain the information necessary to restore the previous version from the current version.

3.1.2 Layers, Granularities and Physical Representations

Versions can be recorded at different layers and granularities. In theory, any granularity in the range between a single byte of a file and the entire file system is conceivable. However, a too fine-grained versioning scheme is likely to create large numbers of versions, which in turn may induce a significant management overhead. On the other hand, a too coarse-grained versioning scheme may be impractical, especially in connection with COW, where it may cause large amounts of unchanged data to be repeatedly copied and thus lead to a waste of I/O resources and storage space.

An obvious way to implement versioning on a conventional block-based file system is to maintain individual versions of blocks and inodes at the internal file system layer. Such an approach is followed by various file systems, including WAFL [HLM94], Episode [CAK+92] and ext3cow [PB05a], which perform COW on data blocks and indirect blocks. Individual versions of a file have their own inodes that reference the respective block versions.

The most widespread alternative to block-level versioning is to maintain versions on a per-file basis, as e.g. done by CFS [GNS88], Wayback [BCD04] and Versionfs [MRWHZ04]. In most cases, per-file versioning schemes are implemented on a superimposed layer above the internal file system layer, which effectively means that a separate file is created for each version. To save disk space and to reduce I/O load, it is often coupled with optimizations: Wayback uses its undo logs to avoid redundancy, whereas Versionfs supports different storage policies that allow for the compression as well as sparse storage of changes.

3.1.3 Storage Locations

Versioning file systems typically store old versions of their data along with the current version on a common storage device, such as a logical volume or a storage pool. Whenever a new version is created, new data is written to the same device and linked to the original data in some way or the other, so as to maintain the chain of versions locally. This approach is found in a range of file systems [BM07, PB05a, HLM94, CAK+92, BCD04, SGSG03].

On block-based versioning file systems, storing new versions along with old versions may lead to an increased degree of fragmentation, as consecutive data blocks of a file may be intermingled with blocks containing data from prior versions. To tackle the problem, *Shrira and Xu* [SX06] suggest to store versions and snapshots on different physical partitions, depending on their expected lifetime. The approach of storing versions and snapshots on separate media is present in a range of other systems [ML04, SX05, XLY+06].

Similarly, *Quinlan* [Qui91] presented an approach that records snapshots on external optical WORM (write-once-read-many) media. Apart from mitigating fragmentation issues, the write-once restriction guarantees immutability of snapshots. Changed blocks of files and directories are initially directed to a disk-backed cache and written to a WORM device when a snapshot is requested.

Aguilera et al. [AKM+07] suggest to combine versioning with hierarchical storage management. Files are spread across multiple storage tiers, which are backed by storage devices with different capabilities in terms of performance and data safety. Aside from being applied to files on the respective tiers, changes are also recorded in a *recovery log*. This log resides on the highest tier with the best recoverability characteristics and provides for a continuous data protection across all tiers.

3.1.4 Version Creation and Retention

Having discussed *how* and *where* versions can be recorded and retained, an issue that remains is *when*. To build a versioning scheme, it is necessary to specify events that trigger the creation and deletion of versions.

Version Creation. Typically, file systems record new versions either with every write or relative to open and close events [MRH09]. The former approach makes it possible to track changes back at the granularity of individual writes and thus allows for a fine-grained tracing of updates over time. However, it generally causes a higher consumption of storage capacity and increases load on the storage devices. Examples of file systems that record versions on every write are CVFS [SGSG03], TierFS [AKM+07], LFS [RO91] and Wayback [BCD04].

Versioning relative to open and close events offers a reasonable compromise between a frequent recording of changes and a low impact on the system. Besides, it ensures that each version reflects a consistent and processable state from an application's point of view, as files can generally be assumed to be in such a state when having been closed after writing. Most file systems that employ such a "close-to-open" versioning policy create new versions upon the first update after a closed file is opened for writing. Prominent examples are the Elephant [SFHV99] file system and Versionfs [MRWHZ04].

Muniswamy-Reddy and Holland [MRH09] examined approaches to record versions based on causal dependencies between accesses rather than individual open, read or write events. In particular, they suggest to track all causal dependencies as a directed graph, where processes and versions constitute the nodes and the data flow between these constitutes the edges. A new version is created each time a new edge i.e., an access of a process to a version, would otherwise induce a cycle in this graph. Such a versioning scheme together with information on causal dependencies allows users and administrators to trace all causal chains of changes back without having to create versions on every write.

As an alternative to the aforementioned approaches, some snapshotting file systems create new versions only after taking snapshots. In general, this approach is more resource-efficient than other ones, as it causes new versions to be created less frequently. File systems that implement such a solution are WAFL [HLM94], which creates snapshots on a regular basis, and AFS [HKM+88], which creates snapshots on demand.

Version Retention. In addition to the question when versions should be created, it is necessary to decide on how long to retain them. While some use cases of versioning may require the complete history of versions to be retained (e.g., post-intrusion diagnosis [SGSG03]), such a comprehensive versioning scheme is not feasible in most cases, as old versions that are rarely accessed are likely to accumulate in the long run and consume a decent share of the available storage space.

A simple solution is hence to restrain the total number of versions. This approach is followed, e.g., by Files-11 [McC90] and CVFS [SGSG03], which only retain a limited number of versions and cause the oldest versions to be discarded when new versions are recorded. Similarly, WAFL [HLM94] limits the total number of snapshots in order to enforce a hard limit on the number of versions.

Log-structured file systems, such as LFS [RO91], typically rely on a cleaner process to dispose of obsolete versions. To reduce the performance impact of the cleaner process on file system workloads, *Blackwell et al.* [BHS95] proposed heuristics to schedule the cleanup activity at times when disks are idle. A trace-based analysis shows that this can be done with up to 97% of all necessary cleanup activity.

The problem of pruning the long-term history of file versions has been further examined by *Santry et al.* [SFH+99]. To retain a history of meaningful versions while keeping the additional storage consumption within reasonable bounds, they suggest a heuristic to detect specific landmark versions. Based on the assumption that users put less emphasis on individual versions if they were created long ago, the heuristic detects groups of old versions that were created during a relatively short time span in the past and only keeps the latest version within each of these groups.

Muniswamy-Reddy et al. [MRWHZ04] presented Versionfs, a file system that allows to combine different policies and to specify both upper and lower bounds for the count, space and life time of versions. Such policies can be defined based on many different properties of files, such as name, extension, size and time of day.

3.2 Metadata Versioning

Aside from file content, versioning also affects metadata. Changes to the metadata of a file or directory, such as updates of ownership information, access rights or timestamps, need to be taken into account as well when it comes to version and snapshot management. Accordingly, metadata versioning is an important aspect of versioning file systems, for which a range of different technical solutions have been developed.

3.2.1 Journaling

A common feature of modern file systems is journaling. Journaling protects a file system from metadata corruption and potentially associated data loss in consequence of system failures and power outages. By batching all metadata updates of a file system operation and recording each batch in a specific log called the *journal*, it is possible to restore the latest consistent state of all metadata. Since the journal records all updates, it

also implicitly records prior versions in the same way as the log-structured approaches described in section 3.1.1. In [SGSG03], *Soules et al.* show that journal-based metadata versioning can reduce the disk space utilization induced by metadata versioning by up to 80% compared to a conventional COW-based versioning scheme.

3.2.2 Versioned B-Trees

Many file systems use index structures to arrange metadata, so as to allow for a fast and efficient retrieval of file metadata and directory contents. A widely used index structure are B-trees [BM72], which guarantee to retrieve, insert and delete any record in up to $O(log(n))$ time, being n the total number of records. A rebalancing of the tree, which occasionally becomes necessary when inserting or deleting records, can be done without significant relocations of data on the underlying storage device.

A *versioned* B-tree [BGO+96] enhances the concept of a B-tree by incorporating multiple versions of a record. A common variant of a versioning B-tree is the *copy-on-write (COW) B-tree*, which finds a use in a range of versioning and snapshotting file system implementations, including Episode [CAK+92], WAFL [HLM94] and ZFS [BM07]. A COW B-tree duplicates all tree nodes up to the root node when a new version is inserted, which generally leads to a rather low metadata update performance and an increased consumption of storage space for versioned metadata. The *multiversion B-tree* proposed by *Becker et al.* [BGO+96] is an optimized alternative to the COW B-tree, which, e.g., is used for the management of directories in CVFS [SGSG03]. It relies on versioned pointers at tree nodes, which obviates the need for COW and thus requires less space. *Twigg et al.* [TBM+11] introduced the *stratified B-tree* as another optimization of the COW B-tree. Stratified B-trees internally rest upon a list of sorted arrays of versioned key-value pairs. They allow for higher insert and range lookup rates at the cost of a slightly decreased lookup rate for individual keys.

3.2.3 LSM-trees

Previous work has also seized the idea of using *log-structured merge trees* (LSM-trees) [OCGO96] for the management of metadata. Although LSM-trees were originally developed as an alternative index structure for transactional database systems, data structures reminiscent of LSM-trees have found their way into various other storage systems. These include Bigtable [CDG+06], Google's storage system for large volumes of structured data, LHAM [MOW98], a hierarchical storage management infrastructure, as well as Rose [SCB08], a storage engine that supports replication. Compared to B-trees, LSM-trees offer particularly high update rates on traditional rotating storage media such as hard drives, as the persistent recording of changes is exclusively done through append writes on the storage device. In particular, changes are written to a log and applied to an in-memory index structure backed by a search tree. When the size of the in-memory index exceeds a threshold, the system triggers a *rolling merge* process, which gradually moves records from the in-memory tree to a persistent on-disk index. This rolling merge process may be I/O intensive, but it can be performed in a completely asyn-

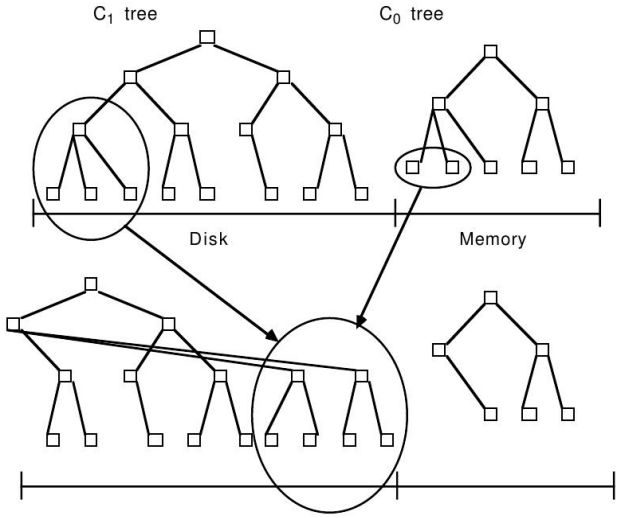

Figure 3.2: Illustration of a two-component LSM-tree, taken from [OCGO96]. The C_1 tree (left) contains a persistent representation of formerly inserted records. The C_0 tree (right) resides entirely in memory and contains records that were recently inserted or updated. When the C_0 tree becomes too large to be held in memory, it is shrunk by gradually merging records into the persistent C_1 tree.

chronous fashion. The principle of a two-component LSM-tree, which consists of an on-disk (C_1) and an in-memory (C_0) component, is illustrated in figure 3.2.

By stacking a new memory-resident C_0 component on top of the existing one and merging each C_0 component together with all lower-level C_0 components and the latest C_1 component into a separate, version-specific C_1 component, it is possible to capture and persistently retain all data stored in an LSM-tree in its current state. BabuDB [SKHH10], a fast and efficient key-value store designed for the management of file system metadata, makes use of this technique to implement snapshots at database level. Further details about BabuDB will be presented in section 3.4. Concepts of BabuDB and LSM-trees were adopted by *Van Heuven van Staereling et al.* [vHvSAvMT11] to implement a management and indexing scheme for file system metadata.

3.3 Management and Versioning of File Content in XtreemFS

As described in section 2.4, XtreemFS builds upon the principle of object-based storage, where file content is managed by OSDs. The efficient handling of files and objects by an OSD is thus essential for the throughput and latency of I/O operations. A core challenge is to provide the ability to quickly retrieve and store objects of a given file. This also involves individual versions of files and objects if versioning is supported.

3.3.1 File and Object Management

Rather than developing a purpose-built storage back-end, we decided to use a local file system for the management of file content in XtreemFS. This eases the development and ensures that the OSD is built upon a mature and well-tested storage subsystem. Widespread object-based file systems follow a similar approach; for instance, Lustre [Clu02] resorts to the local *ext4* file system as the storage back-end across all OSDs.

The internal storage layout of an XtreemFS OSD can be summarized as follows:

- For each XtreemFS file, a directory is maintained on the OSD's local file system. The directory contains all locally stored file content in the form of files.
- Each object version is stored as an individual file. Identifiers for objects and object versions are encoded in the file name. Writing an existing object either triggers a copy-on-write update of the previous version or simply overwrites the object. Which of these two options is chosen depends on the versioning policy, which can trigger version creations on every write or only once within an open-close period.
- In addition to object versions, XtreemFS keeps track of file versions. A file version comprises a specific set of object versions that defines a consistent version of the file's content. Similar to object versions, file versions can be created with every write or when files are closed after writing. Information on file versions is retained in a specific append-only log called *file version log*, which is backed by a file in the local directory of the XtreemFS file. Each time a file version is created, a 128 bit log entry is appended, which consists of two 64 bit integers encoding timestamp and length of the file version.
- XtreemFS resorts to timestamps in order to identify individual versions of files and objects, rather than using logical version numbers. Whenever a new version of a file or object is created, it is bound to a timestamp obtained from the OSD's local clock.

3.3.2 Accessing Object Versions

OSDs maintain a transient open state for each open file, which is initialized with the first `read` or `write` request after opening the file. As trace-based file system analyses have shown that open-close periods of a file often involve multiple subsequent read and write requests [JLA00], such an open state can avoid disk accesses and thus help to speed up the access to open files.

As part of the open state, the OSD maintains an *object version table*, an index structure that allows to efficiently retrieve the local file name for an object version through an object identifier and a timestamp. The object version table is backed by a red-black tree [GS78], which allows for insertions and lookups of object versions in an asymptotically optimal time of $O(log(n))$, being n the total number object versions attached to the file.

Part of the open state is also a memory-resident representation of the file version log, which makes it possible to retrieve file versions repeatedly without potential additional

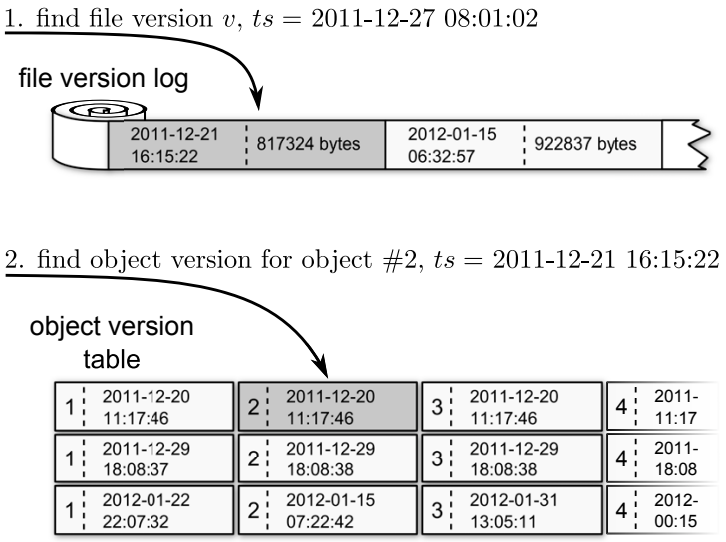

Figure 3.3: Retrieval of an object version. When receiving a `read` request for an object of a specific file version, the OSD first performs a lookup in the memory-resident representation of the file version log, so as to retrieve the correct file version timestamp. In a second step, it retrieves the object version attached to the file version, which is the latest object version with a timestamp smaller than or equal to the file version timestamp.

disk seeks. Since the file version log only undergoes append writes and contains all file versions in the order of their creations, its memory-resident representation can be efficiently backed by a growable array and searched with a binary search algorithm. Figure 3.3 illustrates the retrieval of the correct object version with a `read` request on a snapshot, which involves accesses to the file version log and object version table.

3.4 XtreemFS Metadata Management with BabuDB

The efficient management of metadata is of paramount importance for a file system, as analyses of file system traces have shown that metadata needs to be accessed in response to approximately 75% of all file system calls [JLA00]. Metadata management has particular relevance for object-based file systems, as such file systems often resort to a rather small number of dedicated metadata servers in order to store and maintain metadata of all files.

3.4.1 BabuDB

To provide an efficient storage back-end for metadata in XtreemFS, we developed BabuDB [SKHH10], a key-value store specifically designed for the management of file system

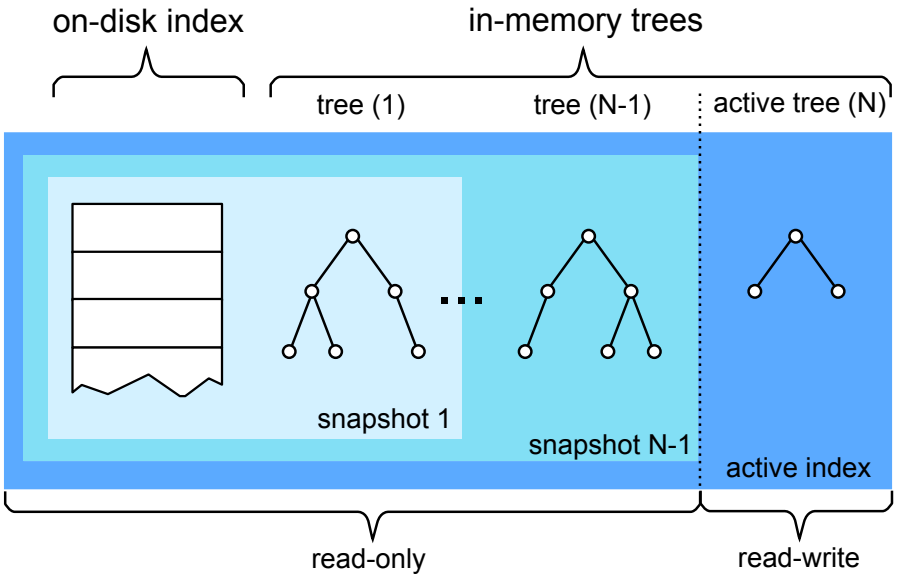

Figure 3.4: BabuDB index architecture

metadata. BabuDB offers various advantages over traditional schemes for the management of file system metadata, most of which resort to B-trees. The most important ones are:

- **Sequential disk writes.** BabuDB exclusively performs sequential disk writes. This rules out costly I/O operations when adding or modifying metadata, such as a re-balancing of persistently stored trees. Since the number of disk seeks is minimized, high throughput numbers can be attained when updating metadata.
- **Atomic operations without locking.** BabuDB supports atomic updates of multiple metadata items without the use of locking mechanisms. This supports the efficient implementation of file system calls like `rename`, which requires an atomic update of multiple metadata items.
- **Efficient management of short-lived files.** BabuDB maintains short-lived files in a particularly efficient manner. Files that are created and deleted within a couple of minutes, which make up about 50% of all files created in a file system [ODCH+85], are likely to vanish before a checkpoint is created, which effectively reduces their creation overhead to a single append write on a log.

At its core, BabuDB is composed of a persistent log and a set of databases, which in turn consist of multiple indices. The log records all kinds of changes, such as updates of key-value pairs and creations or deletions of databases. In the event of a restart of BabuDB, the log is used to recover the latest state of all databases.

Indices allow for a retrieval of individual key-value pairs in $O(log(n))$ time, being n the total number of stored key-value pairs. Their internal design bears resemblance to

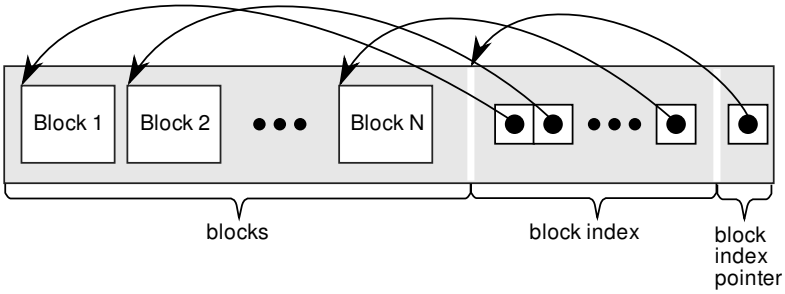

Figure 3.5: BabuDB on-disk index layout

LSM-trees [OCGO96] and the index design of Google Big Table [CDG+06]. As shown in figure 3.4, an index is composed of *in-memory* search trees that contain the latest updates, as well as an *on-disk* index that persistently stores a compacted, efficiently searchable representation the BabuDB index in a former state. In-memory trees are stacked on top of the on-disk index. A lookup starts at the topmost, so-called *active* in-memory tree and goes down the stack to the on-disk index until the respective key is found. Insertions are only performed on the active in-memory tree, which hence reflects the most recent changes.

Database Snapshots. The internal design of BabuDB inherently offers support for snapshots at database level. Taking a snapshot of an index simply creates a new empty in-memory tree on top of the current active tree. Thus, the new tree becomes the active tree, and the previous active tree together with all trees and the on-disk index below reflects an immutable snapshot of the BabuDB index. Lookups on a snapshot start at the corresponding snapshot tree rather than the active tree. This architecture makes it possible to take point-in-time snapshots of arbitrarily large indices without considerably restraining access to the database.

To keep the memory footprint within reasonable bounds, BabuDB initiates a *checkpoint* when the size of the log (and the cumulative size of all in-memory trees across all indices, accordingly) exceeds a configurable threshold. In the course of this, a new snapshot is taken, from which all key-value pairs are retrieved and written to a new on-disk index. The same is done for all user-initiated snapshots, such that each snapshot has a persistent counterpart in the form of an on-disk index. The process of creating a checkpoint may take some time, but it can be executed asynchronously in the background. Finally, the previous on-disk index is replaced with the new one that was created for the checkpoint, all in-memory trees are discarded, and the log is cleaned up by removing log entries that were created prior to initiating the checkpoint.

Internal Data Structures. In-memory trees can be backed by arbitrary search trees. We decided to use red-black trees [GS78] in order to support the insertion and retrieval of key-value pairs in an asymptotically optimal time of $O(log(n))$, similar as for file version tables in the OSD.

The on-disk index is backed by a persistent, immutable list of key-value pairs, which is sorted by keys and searched by means of a binary search algorithm. As shown in figure 3.5, the on-disk index is split up into blocks, where each block contains a fixed number of key-value pairs. The mapping between key ranges and blocks is stored in a separate *block index*, which is initially loaded into memory in order to minimize the number of disk seeks required to find a specific key-value pair. A lookup first searches the block index to retrieve the appropriate block, and queries the block for the key-value pair in a second step. On-disk index files are typically memory-mapped, so as to ensure that frequently accessed key-value pairs remain memory-resident as long as sufficient physical memory is available. Since blocks usually fit in a single memory page, they can be fetched from disk with a single sequential read operation, which only requires a single disk seek in advance.

Range and Prefix Lookups. The ordering of key-value pairs across all in-memory trees and the on-disk index makes it possible to retrieve contiguous ranges of key-value pairs in a particularly efficient manner. Since consecutive key-value pairs can be accessed sequentially without having to perform an individual binary search for each of them, range lookups can be performed in $O(log(n) + k)$ time, being n the total number of key-value pairs and k the number of key-value pairs to be retrieved.

To perform a range lookup on a BabuDB index, BabuDB internally performs individual range lookups across the whole stack of internal indices. The resulting ranges are merged in a similar way as results from a conventional lookup; i.e., the record in the topmost internal index takes precedence if multiple records with the same key exist in different internal indices. A range lookup returns a result set, which allows to iterate over the requested range. Thus, the merging procedure takes place in a stepwise fashion, which keeps the memory footprint low regardless of the number of records in the range.

Aside from common range lookups that return all key-value pairs between a first and a last key, BabuDB also supports *prefix* lookups, which return all those key-value pairs that start with a given key prefix. On the assumption that the ordering of keys is lexicographic, prefix lookups can be internally treated as range lookups, where the range starts with the prefix key, inclusively, and ends with the next larger key of the same length, exclusively.

Atomic Updates. Since BabuDB serializes accesses to an index, multiple key-value pairs can be updated in a single step without additional locking mechanisms. BabuDB provides atomic *insert groups* for this purpose, which may contain multiple updates of different key-value pairs. To ensure atomicity in the face of a crash, each such insert group is persistently recorded by a single entry in the log.

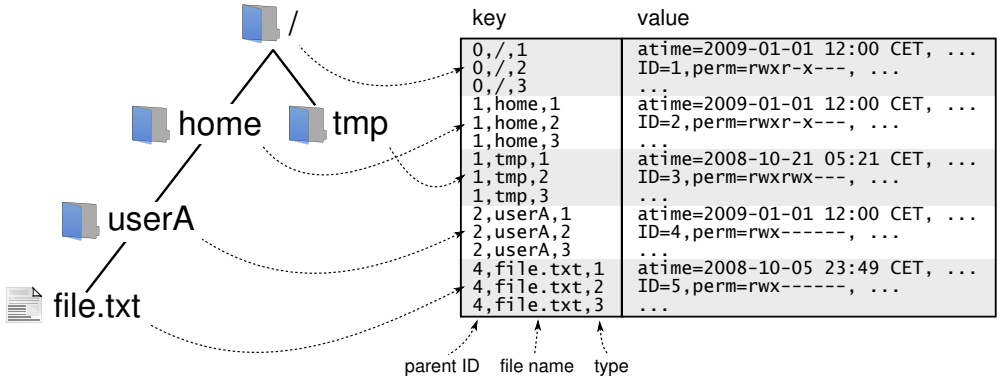

Figure 3.6: BabuDB metadata mapping

3.4.2 Metadata Mapping

The MRC stores and maintains metadata of its volumes in BabuDB. While the metadata of a volume is arranged in a tree-like directory hierarchy, BabuDB only supports flat indices. Accordingly, the MRC needs to translate the tree of files and directories to a set of key-value pairs.

We decided to use keys that consist of three parts: the ID of the file's or directory's parent directory, the name of the file or directory, and a type indicating the frequency of metadata updates. Figure 3.6 illustrates such a representation, which has various advantages:

- Metadata of individual files and directories can be retrieved with a simple prefix or range lookup, where the key is the concatenation of parent directory ID and file name. This minimizes the latency of metadata operations.

- Subdividing the metadata of a file or directory into multiple key-value pairs allows for a separation of frequently and rarely updated items. By using a separate record for items that are affected by frequent operations like `write` (e.g., file sizes and timestamps), the average size of a log entry can be substantially reduced. This increases the maximum throughput of update requests in the MRC.

- The metadata of all files and subdirectories nested in a directory can be retrieved by means of a single prefix lookup, where the key is the parent directory ID. This ensures that directory content can be retrieved in a particularly efficient manner in connection with operations like `readdir`.

- Renaming a file or directory only requires a fixed, small number of key-value pairs to be updated.

3.4.3 Metadata Versioning

To record and maintain versions of metadata, XtreemFS resorts to the inherent ability of BabuDB to take snapshots at database level. A database snapshot can comprise the state of multiple indices and thus record all metadata of an XtreemFS volume in an atomic fashion. As a result, metadata versions reflect the metadata of entire volumes rather than individual files. Although such a versioning scheme limits the flexibility and potential use of the metadata versioning subsystem, it has the advantages of being easy to implement and allowing all metadata of a snapshot to be stored in a separate, contiguous disk region, thus rendering accesses more efficient.

Metadata versions can be captured within a negligible period of time, as creating snapshots at database level essentially only involves the creation of a single empty tree in memory and an append write to the log. Accordingly, recording metadata versions barely affects the responsiveness and availability of the MRC.

3.5 Related Work: Versioning File Systems

Techniques for versioning and snapshots are present in file systems since the late 1970s. Many such file systems have been developed, which are mostly designed to run on a local machine or a single network-attached storage server. A chronological overview is shown in table 3.1.

3.5.1 Local File Systems

Files-11. One of the first versioning file systems is Files-11 [Gol85], the file system of the VMS (and later OpenVMS) operating system. Files-11 constrains the total amount of disk space occupied by different versions by retaining only a limited configurable number of old versions per file. If the maximum number of versions is exceeded, the oldest versions are deleted automatically. Since versioning only applies to files without regard to directories, Files-11 does not support complete snapshots that are aware of preceding changes to the directory structure.

Plan-9 Cached WORM File System. The Cached WORM File System [Qui91], a file system provided with early versions of the Plan 9 operating system in the late 1980s, uses optical WORM (write-once-read-many) media to store file system snapshots. All changes to files and directories are first directed to a cache backed by a disk drive, which allows common read-write access to file system blocks. Changing a file causes a copy-on-write of all previously unchanged blocks to the cache. Taking a snapshot effects that blocks in the cache are synchronously marked for being dumped. A background process writes these blocks to the WORM device and eventually marks the blocks for being copied on write again. Thus, each dump establishes a new snapshot of the file system, which remains stable as it is stored on write-once media.

Year	File System	Characteristic Features
1985	Files-11 [Gol85]	file versioning, remote file access
1988	Cedar [GNS88]	file versioning, sharing of consistent versions
1988	AFS [HKM+88]	scalability, replication, disconnected operations, volume-level snapshots
1991	Cached WORM FS [Qui91]	snapshot storage on optical WORM devices
1992	Episode [CAK+92]	POSIX compliance, fast COW snapshots, transactional metadata updates
1994	WAFL [HLM94]	fast and automatic COW snapshots, fast crash recovery
1996	Spiralog [JL96]	log-structured data storage, on-line backup
1997	Frangipani [TML97]	replication, distributed virtual storage device, distributed snapshots
1999	Elephant [SFHV99, SFH+99]	file and directory versioning, version retention policies
2002	Fossil/Venti [QD02]	archival storage, snapshots, CAS
2002	GPFS [SH02]	parallel SAN access, distributed locking, snapshots
2003	CVFS [SGSG03]	comprehensive versioning for intrusion detection, efficient metadata versioning, multi-version B-trees
2003	ext3cow [PB05a]	copy-on-write, time-shifting interface
2003	Google File System [GGL03]	distribution, replication, atomic appends, snapshots
2004	Versionfs [KK03, MRWHZ04]	stackable, policy-driven version management
2004	Wayback [BCD04]	stackable, undo log
2007	TierFS [AKM+07]	HSM, multi-tier backup recovery, continuous data protection
2007	ROW-FS [CF07]	redirect-on-write, checkpoints
2007	ZFS [BM07]	disk aggregation, RAID-Z, transactions, snapshots

Table 3.1: History of versioning and snapshotting file systems

Fossil. Later versions of Plan 9 came with Fossil[1]. Fossil is an archival file server that stores file system snapshots on Venti [QD02], a content-addressable storage system for archival data. For backup and archiving purposes, Fossil copies whole snapshots to a Venti server. Block versions are assigned to snapshots by means of epoch numbers, which are incremented with each snapshot. Any newly allocated blocks are attached to the current epoch, which makes it easy to determine and copy those block that have changed since the latest snapshot. To reclaim disk space occupied by obsolete snapshots, a lower epoch bound can be specified that helps to identify epoch numbers of versions that do not need to be retained anymore.

Elephant. The Elephant file system [SFHV99] was developed in the late 1990s as a versioning file system for the FreeBSD kernel. Elephant makes use of an *inode log* in order to record metadata versions of each file. In particular, each file version is attached to a distinct inode, which is timestamped on creation in order to allow for a retrieval of prior versions at any given time. Previous versions of data blocks are retained via COW.

One of the unique characteristics of the Elephant file system is its support for different version retention policies [SFH+99]. They define which individual versions of each file should be kept or discarded. Rather than simply disposing of the oldest versions when space on the storage device is reclaimed, it is possible to specify whether to keep all versions of a file, only the latest version, all versions created within a certain time frame or versions that were previously marked as *landmark* versions. To detect landmark versions, the system uses a heuristic that aims to detect groups of old versions that were created during a relatively short time span in the past. Only the latest version in each group will be retained as a landmark version, based on the assumption that users put less emphasis on individual versions if they were created long ago.

CVFS. The Comprehensive Versioning File System (CVFS) [SGSG03] was designed to protect data from intrusions and malicious tampering. It records versions of files and directories on every change, so as to allow for a fine-grained roll-back and tracing of corruptions. As such a comprehensive versioning scheme generates a large number of versions, CVFS especially focuses on an optimized versioning of metadata in order to minimize storage consumption. Unlike conventional approaches that simply duplicate metadata blocks, it relies on a particularly space-efficient journal-based version recording scheme for inodes and indirect blocks, and a multiversion B-tree for directories. Data block changes are recorded in a log. To constrain the number of versions, each version has a limited life span, and expired versions are garbage-collected by a cleaner process.

Ext3cow. Ext3cow [PB05a] is a revised version of the widespread Linux file system ext3 that supports continuous snapshots and versioning. Like Fossil, ext3cow sets a

[1] `http://doc.cat-v.org/plan_9/4th_edition/papers/fossil`

new system-wide epoch number each time a snapshot is taken. In doing so, however, it does not increment a logical counter but assigns a 32-bit integer that reflects the current clock time in seconds since 1970. This way, previous versions can be retrieved by means of time stamps rather than plain numbers. Setting a new epoch number causes any inode, indirect block and data block to be copied only on the first subsequent write, which minimizes the induced overhead.

Versionfs. Versionfs [MRWHZ04, KK03] is a versioning file system that can be stacked on top of any existing file system to add versioning support. Versions are created per file on close. Similar to Elephant, the decision when to purge old versions is policy-driven. VersionFS can either retain a certain number of versions, each version for a predefined period of time, or as many versions as possible according to a space limit for all versions. As opposed to Elephant, VersionFS allows to combine different policies and to specify both upper and lower bounds for the count, space and life time of versions. Policies can be defined based on many different properties of files, such as name, extension, size and time of day.

The approach of a stackable versioning file system is also followed by **Wayback** [BCD04]. A characteristic feature of Wayback is its undo log for each file, which is used to efficiently store prior versions. Any overwritten byte range of a file is appended to the undo log before being replaced, which allows for a fast retrieval of the latest version and for a roll-back to any prior version.

TierFS. TierFS [AKM+07], a file system that integrates multiple storage tiers, combines versioning with hierarchical storage management. It was designed to provide for continuous protection and fast recovery of data. Changes to the file system are not only applied to the appropriate storage tiers but also to a recovery log stored at the highest tier. Through the recovery log, TierFS can take consistent snapshots across multiple tiers without blocking access to the file system, as changes can be confirmed after having been recorded in the log and deferred on the tiers while snapshots are being taken. Log entries, which are recorded at block level, can either be discarded once the corresponding changes have been applied to the tiers, or kept for continuous data protection until a backup has been created.

ZFS and Modern Versioning File Systems. In 2005, ZFS [BM07] was released as one of the most advanced local file systems. Aside from snapshots and versioning, ZFS offers a wide range of sophisticated features, such as aggregating multiple physical disks to a large virtual storage device, RAID and checksums for data safety, data compression and deduplication for an efficient usage of storage capacity, encryption, transactional modifications of multiple files and directories, and various others. Snapshot support is based on a COW scheme that works similar to the one used in ext3cow. However, it performs a COW of all blocks and indirect blocks with each transaction, which is required already to ensure transactional semantics.

Aside from ZFS, various other snapshotting file systems have been developed recently. Btrfs[2] aims to provide the functionality of ZFS through an open-source implementation based on multiversion B-trees. The idea of Tux3[3] is to minimize versioning overhead by efficiently recording changes. NILFS[4] is a log-structured file system that supports snapshots and fast crash recovery. Next3[5] is a commercially distributed derivative of the ext3 file system that supports snapshots for consistent backups of cloud storage systems. NTFS, the standard file system for modern Windows distributions, comes with the Volume Shadow Copy Service, which preserves former versions at block level subsequent to a volume snapshot.

3.5.2 Network File Systems

CFS. One of the first file systems that support versioning is the Cedar File System (CFS) [GNS88], a remote multi-user file system developed in the early 1980s. The initial goal of CFS was to provide a tool that facilitates the sharing of consistent versions of software modules between programmers. To provide for a consistent versioning scheme, all files are immutable, which means that any change to a file causes a new version of the file to be created. File versions are created locally and copied to the server for sharing afterwards. Since file versions are sequentially numbered and timestamped, arbitrary prior versions can be identified and accessed. However, CFS does not support snapshots, as versioning is restricted to files without involving directories and other kinds of metadata.

AFS. The Andrew File System (AFS) [HKM+88] is a scalable multi-user file system that runs across a potentially large set of hierarchically organized servers. Files and directories are arranged in volumes, each residing entirely on a single server. For the purpose of creating consistent backups on tape, servers support snapshots at volume level. Taking a snapshot creates a clone of a volume, which consists of hard links to the original data. AFS uses copy-on-write to protect the original data from being changed.

Episode and WAFL. As an alternative to AFS, the Episode file system [CAK+92] was developed in the early 1990s. Episode also employs COW techniques to conserve prior versions of files and directories in the face of changes. So does Write Anywhere File Layout (WAFL) [HLM94], a file system for storage appliances developed by NetApp. In accordance with the typical design pattern of a local Unix file system, Episode and WAFL internally rely upon a tree of disk blocks containing inodes and data. Taking a snapshot in WAFL causes the root inode to be duplicated, which effects a subsequent COW of any changed block and other blocks on the path between the root inode and

[2] `https://btrfs.wiki.kernel.org`
[3] `http://tux3.org/`
[4] `http://www.nilfs.org`
[5] `http://www.ctera.com/home/next3.html`

the changed block. Episode also employs COW to protect data blocks from being overwritten but actively clones all inodes when taking a snapshot. This leads to a lower overhead in writing a block at the cost of a higher overhead in creating snapshots. To reduce the increased write overhead, WAFL collects `write` requests in a cache and applies them to the storage device as a batch.

Spiralog. Spiralog [JL96], a cluster file system that was developed in the mid 1990s as a successor of Files-11 for the OpenVMS operating system, follows an entirely different approach to the management of versions and snapshots. The file system stores files and directories on a server backed by a log-structured file system (LFS), which exclusively records updates in a system-wide append-only log. Since no previous log entries are overwritten, the log contains the history of all previous changes and hence all previous states. Taking a snapshot only requires to adequately mark the most recent log entry.

To quickly access files and directories without traversing the entire log in order to recover the latest version, the LFS uses an additional B-tree [WBW96]. The B-tree maps byte ranges in the virtual address space of the underlying device to those log entries containing the most up-to-date version. Thus, data can be read at the approximate speed of a B-tree based file system. Writing is particularly efficient, as only it involves append writes to the log. To free space once the log becomes full, a cleaner compacts the log and disposes of obsolete log entries that do not contain data in use and are not attached a snapshot.

ROW-FS. ROW-FS [CF07] is a version-aware file system designed for the management virtual machine images in a grid computing environment. The file system is stacked on top of a mounted NFS file system on the client side, where it intercepts all modifications and records them on a local shadow server using redirect-on-write. This approach makes it possible to run multiple instances of the same virtual machine, where each instance has its own data. To avoid frequent redirects when reading data, ROW-FS makes use of client-side caching.

3.5.3 Distributed File Systems

Frangipani. Frangipani [TML97] appeared as one of the first file systems that support snapshots across multiple physically distributed data partitions. Frangipani is composed of a set of distributed servers that run on top of the Petal virtual disk system [LTT96]. Petal supports versioning at the level of virtual disk blocks. Similar to Fossil, snapshots cause a global epoch number to be incremented, which in turn causes future block versions to be copied on write. To ensure that snapshots are consistent despite the fact that multiple servers are involved, a lock inhibits access to the system while snapshots are triggered across the servers.

Google File System. In the early 2000s, Google's need for a storage infrastructure that allows for a fast processing of large data volumes lead to the development of the Google

File System [GGL03]. The Google File System was designed to run across multiple racks on a local cluster. It consists of a master node, which effectively acts as a metadata server, and a multitude of *chunk servers* that store file content in chunks of the same size. Snapshots, which can be taken at file and directory level, heavily rely on locking mechanisms. When a snapshot is requested, the master first revokes all locks across all chunk servers, which causes outstanding write requests to be deferred, and then duplicates all affected metadata. The first write to a chunk after a snapshot causes the chunk to be duplicated.

GPFS. GPFS [SH02] is one of the most widely used parallel cluster file systems. GPFS allows for a consistent parallel access to a SAN through a file system interface. Similar to the Google File System, GPFS supports consistent snapshots by means of locking and block-level COW.

4 Snapshot Consistency in Distributed File Systems

The management of versions described in the previous chapter is essential to freeze and retain prior states of the file system. To define a meaningful concept of snapshots, however, it is also necessary to specify the selection of individual versions of different data items that constitutes a snapshot.

A snapshot is generally considered as *consistent* if all versions have in common that they appear to be captured simultaneously [Lyn96] i.e., reflect a state in which they were at the same instant. In distributed file systems, however, there is no external observer who can record all versions across all servers at a single point in time. Instead, servers have to individually record their local versions when a snapshot is taken. Unfortunately, there is no means of doing this at the same instant without a common global time or perfectly synchronized clocks.

To illustrate the problem of snapshot consistency, Chandy and Lamport use the allegory of a group of photographers observing a panoramic, dynamic scene [CL85]. The goal of these photographers is to take a picture that reflects the overall scene at a scheduled point in time. Since the scene is too large to be captured by a single photographer, all photographers have to take individual pictures of different sections, which will later be put together. However, the group of photographers cannot push the releases of their cameras at the exact same point in time, nor can they influence the scene, e.g., by freezing the locations of all objects while taking the pictures.

As a result, it is necessary to take relaxed snapshot consistency models into consideration. These have to offer meaningful alternatives to the strict semantics of a common global point in time at which the individual snapshots are taken, while still being in line with the expectations of users and applications.

This chapter formally introduces and describes three different snapshot consistency models for file systems and analyzes their semantics and properties. It starts with a formal model of a distributed file system (4.1) and a formal definition of a snapshot (4.2), based on which it presents the consistency models (4.3) and analyzes their individual properties (4.4). Finally, it provides an overview of previous work in the area of distributed snapshots and snapshot consistency (4.5) and summarizes the results (4.6).

4.1 System Model

Our system model is based on the *timed asynchronous model* of a distributed system defined in [CF99]. The timed asynchronous model aims to provide a more suitable description of a real-world distributed system than the common time-free model [FLP85],

in that it incorporates the concept of real time. More specifically, it assumes that processes are equipped with real-time clocks, which give them the ability to observe events in a temporal context and to detect failures based on message timeouts. Our model covers crash-stop and crash-recovery failures but no byzantine [LSP82] failures.

4.1.1 Distributed File System

We regard a distributed file system DFS as an undirected graph with a set of hosts H as vertices and a set of network links L as edges:

$$DFS := (H, L) \tag{4.1}$$

The set of hosts H comprises a set of *clients* C and a set of *servers* S:

$$H := S \cup C \tag{4.2}$$

We distinguish between clients and servers because of their different roles and properties. Clients are only temporarily connected to the network. They may run on any machine with access to the network and may join and leave the system at any time. As a consequence, they are anonymous and invisible until they connect to a server. They can communicate with servers but not with each other. We further assume that clients do not hold any persistent state that needs to be captured in a snapshot. All such state is held by the servers, which, barring failures, are permanently connected and reachable. This reflects the conditions of the most common architectures of distributed file systems, like NAS, SAN and object-based storage.

Communication. Any two hosts with connecting network links have the ability to exchange messages with each other. As illustrated in figure 4.1, we assume that there are network links between any two servers, as well as any client-server pair:

$$L := \{\{s_1, s_2\} : \ s_1, s_2 \in S \wedge s_1 \neq s_2\} \cup \{\{s, c\} : \ s \in S \wedge c \in C\} \tag{4.3}$$

Communication relies on the exchange of messages along network links. We model the sending and receipt of messages as *events* occurring on the respective hosts. Similar as in the timed asynchronous model, each successfully delivered message is bound to a *send* and *receive* event. A *send* or *receive* event can be regarded as a tuple (h, t), where h describes the host and t the point in real time at which the event occurred. We define two functions `send` and `recv` that deliver the *send* and *receive* events for a message m. The model is illustrated in figure 4.2.

Definition 4.1 (Message)**.** *A **message** $m = (s, r)$ is defined by its **send** event $s \in H \times \mathbb{R}$ and its **receive** event $r \in H \times \mathbb{R}$. Being $M_{all} \subset (H \times \mathbb{R}) \times (H \times \mathbb{R})$ the universe of all messages successfully exchanged between hosts, $\texttt{send} : M_{all} \to H \times \mathbb{R}$ returns the send event, and $\texttt{recv} : M_{all} \to H \times \mathbb{R}$ returns the receive event of a message.*

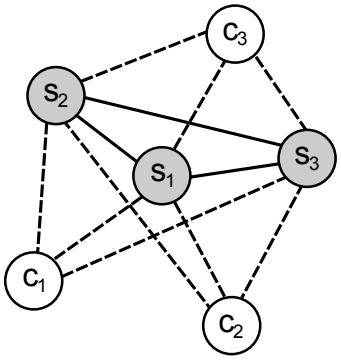

Figure 4.1: Graph-based illustration of a distributed file system. Gray circles refer to servers interconnected through permanent network links (solid lines), white circles refer to clients with temporary network links (dashed lines).

We assume that any successfully exchanged message is delivered within a finite period of time after having been sent, which is greater than zero:

$$\forall((h,t),(h',t')) \in M_{all}\ (t < t') \tag{4.4}$$

4.1.2 File System State

The state of a file system is defined by the state of all its files and directories, including their contents and metadata. In a distributed file system, this state is spread across all servers $s \in S$, such that each individual server holds a part of the global state. We use a generic model of server state that abstracts from the characteristics of individual server types.

State-Changing Events. To lay the foundation for a formal definition of snapshots, it is necessary to take into account how server state changes over time. We assume that changes to server state occur in response to specific events on the servers, so-called *state-changing events*. Similar to a communication event, such an event is bound to a server and a point in time at which it occurs.

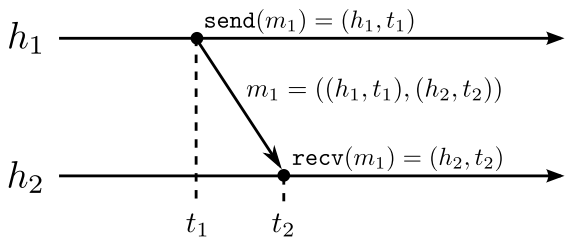

Figure 4.2: Illustration of a message and the associated *send* and *receive* events

Definition 4.2 (State-changing event)**.** *A **state-changing event** causes a change in the state of a server. Being $\mathcal{S}_{all} \subset S \times \mathbb{R}$ the universe of all state-changing events on all servers, the tuple $(s,t) \in \mathcal{S}_{all}$ describes a **state-changing event** that occurs on a server s at the point in time t.*

For our theoretical considerations, we assume that each state-changing event occurs at a precise instant of time, state changes are atomic, and different state-changing events on a server occur at different times. In practice, these requirements can be fulfilled by enforcing a total processing order on events.

State-changing events correspond to those events that trigger the creation of new versions, as discussed in section 3.1.4. They determine the granularity at which state transitions are perceived and states are captured in snapshots. Accordingly, a state-changing event on a storage server can, e.g., be the receipt of a `write` request in order to represent a "version on every write" policy, or the receipt of a `close` request in order to represent a "close-to-open" policy.

Server State. The state of a server is defined by the history of of all state-changing events on the server.

Definition 4.3 (Server state)**.** *The function $\texttt{state}_s : \mathbb{R} \rightarrow \mathcal{P}(\mathcal{S}_{all})$ returns the **state of a server** $s \in S$ at a certain time. It is defined by the set of all state-changing events $\{(s,t_1),\ldots,(s,t_n)\} \subseteq \mathcal{S}_{all}$ that occurred on s up to the time t:*

$$\texttt{state}_s(t) := \{(s,t') \in \mathcal{S}_{all} : \; t' \leq t\}$$

Given that versions are timestamped, we assume that the version management ensures that a server can determine its local state $\texttt{state}_s(t)$ at any point in time t.

4.1.3 Time and Clocks

In accordance with the timed asynchronous model, we assume that each host $h \in H$ is equipped with a reasonably precise real time clock that is capable of generating fine-grained timestamps. We model such a host clock as a function that maps real time onto clock time.

Definition 4.4 (Host clock)**.** *The function $c_h : \mathbb{R} \rightarrow \mathbb{N}$ represents the **clock of a host** $h \in H$. $c_h(t)$ returns a timestamp that reflects the local time on h at the point in time t.*

Our model uses natural numbers to represent timestamps, as the underlying hardware clocks generate timestamps in a discrete rather than a continuous range. A timestamp reflects the time at a well-defined precision that has elapsed since the beginning of a well-defined epoch, such as nanoseconds since January 1st, 1970, 12am UTC.

We assume that host clocks are correct according to the timed asynchronous model, which implies that they display strictly monotonically increasing values. Although host clocks proceed in discrete steps, we stipulate that their granularity is fine enough to ensure that two subsequently generated timestamps always differ from each other:

$$\forall h \in H, t_1, t_2 \in \mathbb{R} \; (t_1 > t_2 \Leftrightarrow c_h(t_1) > c_h(t_2)) \tag{4.5}$$

Clock Drift. To ensure a meaningful relationship between clock time and real time, we further assume that host clocks run within a narrow, linear envelope of real time, as specified by the timed asynchronous model. Accordingly, clock drift rates are bounded by a fixed, system-wide ρ, with $0 < \rho \ll 1$:

$$\forall h \in H, t \in \mathbb{R}, \delta \in \mathbb{R}_0^+ \ (c_h(t) + (1-\rho)\delta \leq c_h(t+\delta)) \tag{4.6}$$

In practice, computers measure time by counting oscillations of a battery-backed hardware clock based on a quartz oscillator [Mil03]. Such oscillations recur at a fairly constant rate, which typically varies by less than 0.1 parts per million [RV10] (i.e., $\rho < 10^{-7}$), and hence leads to a clock drift of no more than a few milliseconds per day.

4.1.4 Summary

The most important symbols defined in this section are summarized in table 4.1.

H	set of all hosts
S	set of all servers
C	set of all clients
M_{all}	set of all messages successfully exchanged between hosts
(h, t)	event occurring on a host h at time t
$\texttt{send}(m)$	*send* event of a message m
$\texttt{recv}(m)$	*receive* event of a message m
$\mathcal{S}_{all}$	set of all state-changing events on all servers
$\texttt{state}_s(t)$	state of a server s at time t
$c_h(t)$	clock time on host s at time t
ρ	upper bound for the drift rate of host clocks

Table 4.1: Symbols

4.2 Global States and Snapshots

Considering the initially described challenge of snapshot consistency, a snapshot ought to reflect a global file system state that is as close as possible to the momentary state when the snapshot was taken. Such a global state is composed of local states of all servers. Accordingly, a snapshot can be formally defined as an aggregate of different server states.

4.2.1 Definition

We define a snapshot through a set of consecutive state-changing events that aggregates certain states of all servers.

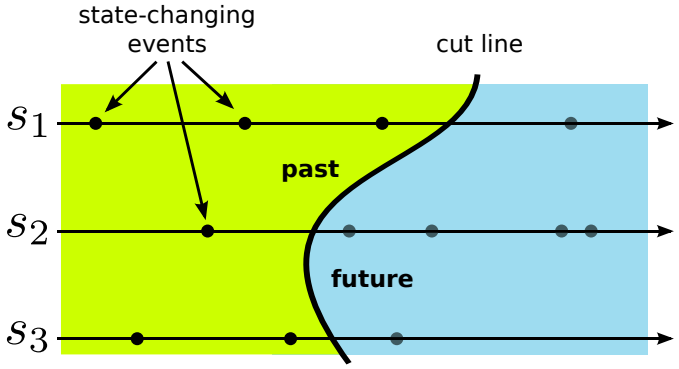

Figure 4.3: Cut representing a global state of a distributed system

Definition 4.5 (Snapshot)**.** *A **snapshot** $\mathcal{S} \subseteq \mathcal{S}_{all}$ is a subset of all state-changing events that occurred up to a specific point in time on each server. It fulfills the following condition:*

$$(s,t) \in \mathcal{S} \ \wedge \ \exists (s,t') \in \mathcal{S}_{all} \ (t' < t) \ \Rightarrow \ (s,t') \in \mathcal{S}$$

Our concept of a snapshot is equivalent to the concept of a *cut* introduced in [KT87], which splits up the set of all events in a *past* and a *future* subset (see figure 4.3). Restrictions concerning the exact selection of events i.e., the behavior of the cut line, depend upon the associated consistency model, which will be discussed in section 4.3.

4.2.2 Snapshot Events

To capture a global state of a distributed file system, each server has to record its individual state. We assume that the process of capturing all server states is triggered by an initial *snapshot event* that occurs on one of the servers. Such an event can, e.g., be an internal timer event if snapshots are taken at regular intervals, or the delivery of a snapshot request sent by a client.

4.3 Snapshot Consistency Models

In [Lyn96], **Lynch** provides a generic definition of snapshot consistency in distributed systems. According to this definition, a snapshot of a distributed system that comprises multiple distinct processes is *consistent* if

> "[...] it looks to the processes as if it were taken at the same instant everywhere in the system".

Accordingly, it is necessary to specify the circumstances under which the processes (which correspond to *servers* in our system model) regard an instant as the same.

4.3.1 Ordering Events

In order to formally specify the guarantees and properties of a consistent snapshot, it is necessary to adequately define the concept of an *instant* in consideration of the fact that local clocks are the only timekeepers servers have access to. An instant is only meaningful if it is bound to an observable event in the system, such as the delivery of a message or the occurrence of a state change. To render instants comparable, it is hence necessary to define a temporal ordering on events, which has to apply to all events in the system. Such an ordering specifies for any two events whether one happened before i.e., at an earlier instant than the other one, or both events happened concurrently i.e., at the same instant.

The "Happened Before" Relation. Hosts are capable of ordering local events according to the local clock times at which they occur. For any two local events e and e', a host can determine if either e happened before e' or e' happened before e. Lamport referred to this as the *"happened before"* relation on a single process [Lam78]. We adopt Lamport's '$\rightarrow$' notation to formally denote this relation.

Definition 4.6 (Single-host "happened before" relation)**.** *An event* (h, t) ***happened before*** *an event* (h, t') ***on a host*** h *iff it occurred earlier than* t' *according to* h*'s local clock:*

$$(h, t) \rightarrow (h, t') :\Leftrightarrow c_h(t) < c_h(t')$$

It is obvious that the "happened before" relation on a single host only defines a partial ordering of events, as it cannot be applied to events that occurred on different hosts. To define a temporal relationship between arbitrary events, it is necessary to extend the relation to a total ordering that applies to any two events in the system.

4.3.2 Point-in-Time Consistency

The most obvious way of extending the "happened before" relation to a total ordering is to simply extend the domain, such that the definition applies to all hosts in the system. To provide for a meaningful concept of snapshots, however, this requires that timestamps generated by different server clocks be comparable in a meaningful manner.

Global Time. A restrictive requirement that renders clock timestamps comparable throughout all servers is the existence of a *global time*. This effectively means that all server clocks run perfectly in sync with each other:

$$\forall t \in \mathbb{R}, s, s' \in S \; (c_s(t) = c_{s'}(t)) \tag{4.7}$$

On that condition, it is safe to say that an event happened before another event if it occurred earlier according to any server's clock. We use the symbol '$\overset{g}{\rightarrow}$' to denote such a total ordering of events based on a global time.

Definition 4.7 ("Happened before" relation based on global time)**.** *An event* (s,t) ***happened before*** *an event* (s',t') ***on the supposition of a global time****, denoted by* $(s,t) \overset{g}{\rightarrow} (s',t')$*, iff it occurred earlier than* t' *according to all servers' local clocks:*

$$(s,t) \overset{g}{\rightarrow} (s',t') :\Leftrightarrow \forall s'' \in S\ (c_{s''}(t) < c_{s''}(t'))$$

The "happened before" relation does not define a relationship between events that happened at the same time on different servers. We define such events as *concurrent*.

Definition 4.8 (Concurrency relation based on global time)**.** *Two events* (s,t) *and* (s',t') *happened* ***concurrently on the supposition of a global time****, denoted by* $(s,t) \parallel_g (s',t')$*, iff* (s,t) *and* (s',t') *happened at the same point in time according to any server's local clock:*

$$(s,t) \parallel_g (s',t') :\Leftrightarrow \forall s'' \in S\ (c_{s''}(t) = c_{s''}(t'))$$

Point-in-Time Snapshots. The existence of a global time across all servers ensures that all servers see the same ordering of events. Based on this ordering, it is possible to define the concept of *point-in-time (consistent)* snapshots.

Definition 4.9 (Point-in-time consistent snapshot)**.** *A snapshot* $\mathcal{S} \subseteq \mathcal{S}_{all}$ *is* ***point-in-time consistent*** *with respect to a snapshot event* (s,t_0)*, denoted by the predicate* $cons_g(\mathcal{S},(s,t_0))$*, if it comprises all state-changing events across all servers that happened before or concurrently with* (s,t_0) *under the terms of a global time:*

$$cons_g(\mathcal{S},(s,t_0)) :\Leftrightarrow\ \mathcal{S} = \{(s',t') \in \mathcal{S}_{all} :\ (s',t') \overset{g}{\rightarrow} (s,t_0) \vee (s',t') \parallel_g (s,t_0)\}$$

4.3.3 Causal Consistency

A practical snapshot consistency model needs to be built upon a relaxed timing model that observes time from a local server's point of view. A common approach to define such a model of time is to observe potential causal dependencies between events.

Causal Dependencies. In accordance with the nature of cause and effect, the causal precedence of one event over another one implies its temporal precedence. If it is safe to say that the occurrence of an event e has or may have caused the occurrence of an event e', e must have happened before e'. Lamport suggested to extend the single-host "happened before" relation to a total ordering by monitoring causal dependencies between events on different hosts [Lam78]. The obvious way of doing this is to observe the exchange of messages.

The event of sending a message causally precedes the event of receiving the message. This complies with assumption (4.4), which states that a message received by a host must have been sent earlier by some other host. We use the symbol '$\xrightarrow{c}$' to denote causal precedence of events:

$$\forall m \in M_{all}\ (\texttt{send}(m) \xrightarrow{c} \texttt{recv}(m)) \tag{4.8}$$

Furthermore, causal dependencies can be assumed between events that occurred on the same host. An event (h,t) causally precedes an event (h,t') on a host h iff (h,t) happened before (h,t') according to the "happened before" relation on a single host:

$$(h,t) \xrightarrow{c} (h,t') \Leftrightarrow (h,t) \rightarrow (h,t') \tag{4.9}$$

In conformity with Lamport's definition, the causality-based "happened before" relation can thus be formally defined as the transitive closure of the "happened before" relation on a single host and the "happened before" relation derived from the causal dependencies of message exchanges between hosts.

Definition 4.10 ("Happened before" relation based on causal dependencies)**.** *An event (h,t)* ***happened before*** *an event (h',t')* ***on the supposition of causal dependencies*** *if either $h = h'$ and $(h,t) \rightarrow (h',t')$, or if (h,t) is equal to or happened before the* send *event of a message m, and the corresponding* receive *event is equal to or happened before (h',t'):*

$$(h,t) \xrightarrow{c} (h',t') :\Leftrightarrow \begin{cases} (h,t) \rightarrow (h',t'),\ \ h = h' \\ \exists m \in M_{all} \\ \quad (((h,t) = \texttt{send}(m) \vee (h,t) \xrightarrow{c} \texttt{send}(m)) \wedge \\ \quad ((h',t') = \texttt{recv}(m) \vee \texttt{recv}(m) \xrightarrow{c} (h',t'))),\ \ h \neq h' \end{cases}$$

Two distinct events without a (potential) causal relationship are regarded as *concurrent*. For concurrent events, it is impossible to determine a temporal ordering.

Definition 4.11 (Concurrency relation based on causal dependencies)**.** *Two events (h,t) and (h',t') are* ***concurrent on the supposition of causal dependencies****, denoted by $(h,t) \parallel_c (h',t')$, if neither (h,t) happened before (h',t') nor (h',t') happened before (h,t):*

$$(h,t) \parallel_c (h',t') \ :\Leftrightarrow\ \neg((h,t) \xrightarrow{c} (h',t') \vee (h',t') \xrightarrow{c} (h,t))$$

Figure 4.4 illustrates the relationships of causally dependent and concurrent events.

Causally Consistent Snapshots. The "happened before" relation for causally-related events effectively orders events by the times of their occurrence. *Mattern* used the term *virtual time* [Mat89] to describe the resulting model of time. Under the terms of this model, a snapshot is *causally consistent* if all comprised state-changing events either causally precede the snapshot event or happened concurrently with it.

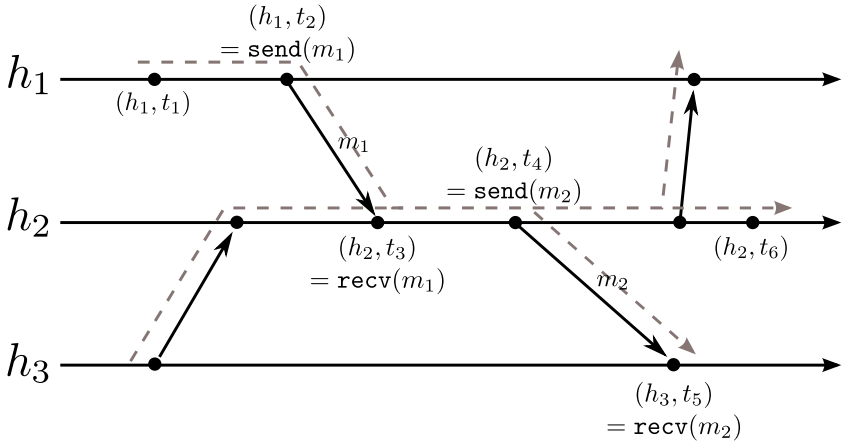

Figure 4.4: Causal dependencies between events. Each host (i.e., h_1, h_2, h_3) has its own timeline at which events occur at different times (i.e., $t_1, \ldots, t_6$). Black solid arrows indicate messages exchanged between hosts, gray dashed ones visualize the resulting graph of causal dependencies between events. Examples of some true statements are: $(h_1, t_1) \xrightarrow{c} (h_1, t_2)$, $(h_1, t_2) \xrightarrow{c} (h_2, t_4)$, $(h_1, t_1) \xrightarrow{c} (h_3, t_5)$, $(h_2, t_6) \parallel_c (h_3, t_5)$.

Definition 4.12 (Causally consistent snapshot)**.** *A snapshot $\mathcal{S} \subseteq \mathcal{S}_{all}$ is **causally consistent** with respect to a snapshot event (s, t_0), denoted by the predicate $\texttt{cons}_c(\mathcal{S}, (s, t_0))$, if it contains all state-changing events that happened before (s, t_0) but no state-changing events before which (s, t_0) happened, under the terms of a causality-based "happened before" relation:*

$$\texttt{cons}_c(\mathcal{S}, (s, t_0)) :\Leftrightarrow \mathcal{S} \supseteq \{(s', t') \in \mathcal{S}_{all} : (s', t') \xrightarrow{c} (s, t_0)\} \wedge \nexists (s', t') \in \mathcal{S}\ ((s, t_0) \xrightarrow{c} (s', t'))$$

A causally consistent snapshot reflects a global state that could have existed at the instant of the snapshot event, as it guarantees that each event is captured together with the entire graph of causally precedent events. To describe such a snapshot, *Koo and Toueg* introduced the concept of a *consistent cut* [KT87]. Figure 4.5 illustrates the difference between consistent and an inconsistent cuts.

4.3.4 Consistency Based on Synchronized Clocks

Observing causal dependencies between events helps to define a relaxed yet meaningful "happened before" relation, which is based upon a less restrictive model of time and snapshot consistency. An alternative approach to attain this goal is to define a real time-based "happened before" relation that relaxes the concept of a global time. On the assumption that server clocks are loosely synchronized, it is possible to capture individual server states within a narrow time frame.

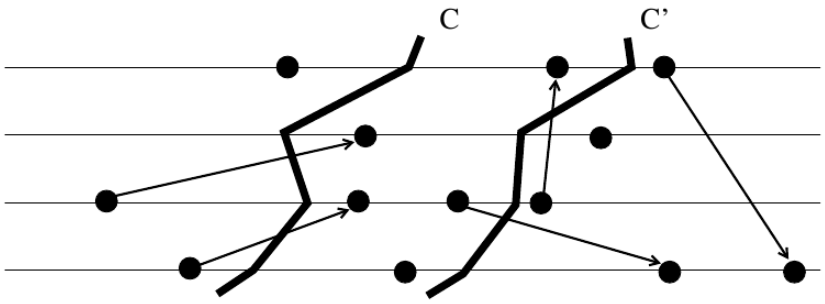

Figure 4.5: Consistent (C) and inconsistent (C') cuts, taken from [Mat93]. A cut is consistent if no messages received in the past were sent in the future.

Loosely Synchronized Clocks. Subject to the inherent imprecision of hardware timekeepers, host clocks tend to drift apart over time [CF99]. Accordingly, it is necessary to bound clock drift in order to make sure that clock times of different servers can be compared in a meaningful manner. We thus assume that server clocks be internally synchronized [Cri89] by a maximum deviation of ϵ; this means that at any point in time t, an upper bound ϵ is known for the clock drift between any two servers s and s':

$$\forall s, s' \in S, t \in \mathbb{R} \; (|c_s(t) - c_{s'}(t)| \leq \epsilon) \tag{4.10}$$

The practical aspects and limitations of clock synchronization in a network have been extensively studied in literature. Clock synchronization protocols like NTP [Mil91, MMBK10], PTP [Ins02] and RADclock [RV10] keep clocks in a distributed system in sync with an external reference clock. On a wide area network like the Internet, such protocols can limit the drift between two clocks to a range between milliseconds and nanoseconds [Mil03]. Alternatively, GPS receivers can synchronize local clocks with GPS satellite timekeepers, which allows for an accuracy in the range of microseconds [PV02].

Relaxed Global Time. With loosely synchronized clocks, the concept of a global time can be approximated. On the assumption that an upper bound on the maximum clock drift is known as specified in (4.10), we define a "happened before" and concurrency relation that is solely based upon timestamps generated by server clocks.

Definition 4.13 ("Happened before" relation based on synchronized clocks)**.** *An event* (s, t) ***happened before*** *an event* (s', t') ***on the supposition of loosely synchronized clocks****, denoted by* $(s, t) \xrightarrow{l} (s', t')$*, iff it occurred earlier on the clock of* s *than* (s', t') *did on the clock of* s'*:*

$$(s, t) \xrightarrow{l} (s', t') :\Leftrightarrow c_s(t) < c_{s'}(t')$$

Definition 4.14 (Concurrency relation based on synchronized clocks)**.** *Two events* (s, t)

and (s', t') ***happened concurrently on the supposition of loosely synchronized clocks****, denoted by* $(s, t) \parallel_l (s', t')$*, if* (s, t) *and* (s', t') *happened at the same time according to the local clocks of* s *and* s'*:*

$$(s, t) \parallel_l (s', t') :\Leftrightarrow c_s(t) = c_{s'}(t')$$

The "happened before" relation for loosely synchronized clocks defines the ordering of events as seen from the perspective of an individual server. Such an ordering may deviate from the ordering observed by a different server. Figure 4.6 shows an example that illustrates the different viewpoints of individual servers.

Snapshots based on Synchronized Clocks. Based on this "happened before" relation, we define a real time-based consistency model for snapshots that relaxes the constraints of point-in-time consistency.

Definition 4.15. *A snapshot* $\mathcal{S} \subseteq \mathcal{S}_{all}$ *is* ***consistent*** *with respect to a snapshot event* (s, t_0) ***on the supposition of loosely synchronized clocks****, denoted by the predicate* $\texttt{cons}_l(\mathcal{S}, (s, t_0))$*, if it captures all state-changing events on each server* s' *that happened before or concurrently with* (s, t_0) *according to the local clocks of* s' *and* s*:*

$$\texttt{cons}_l(\mathcal{S}, (s, t_0)) :\Leftrightarrow \mathcal{S} = \{(s', t') \in \mathcal{S}_{all} : (s', t') \xrightarrow{l} (s, t_0) \vee (s', t') \parallel_l (s, t_0)\}$$

Despite each server having its individual view on the temporal ordering of events, there is an upper bound on the time frame during which the state captured in a snapshot depends upon the effective relative drift of all server clocks. We will now show that the ordering of events as observed by the servers may only differ during a time span of approximately ϵ, which contains the point in time t_0 at which the snapshot was taken.

Lemma 4.1. *A snapshot* $\mathcal{S}$ *with* $\texttt{cons}_l(\mathcal{S}, (s, t_0))$ *contains all state-changing events* (s', t') *that occurred up to the time* $t_0 - \frac{1}{1-\rho}\epsilon$*:*

$$\texttt{cons}_l(\mathcal{S}, (s, t_0)) \wedge \exists (s', t') \in \mathcal{S}_{all} \; (t' \leq t_0 - \frac{1}{1-\rho}\epsilon) \Rightarrow (s', t') \in \mathcal{S}$$

Proof. Let us assume that a state-changing event $(s', t') \in \mathcal{S}_{all}$ exists with $t' \leq t_0 - \frac{1}{1-\rho}\epsilon$. On s''s local clock, (s', t') occurred at $c_{s'}(t')$. As claimed in (4.10), server clocks are loosely synchronized, which implies that $|c_{s'}(t') - c_s(t')| \leq \epsilon$. Consequently, the following condition is satisfied:

$$c_{s'}(t') \leq c_s(t') + \epsilon \tag{4.11}$$

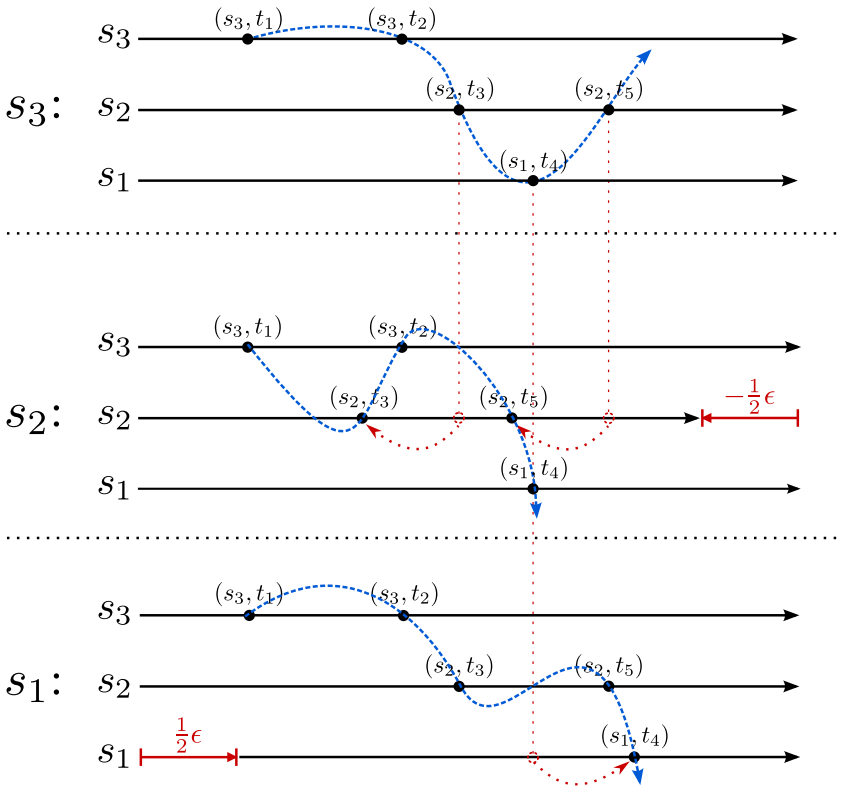

Figure 4.6: Different temporal orderings of events under different clock drifts. The timeline diagrams illustrate three different views to the same sequence of events, as seen by each of the three servers s_1, s_2, and s_3. s_1 has a clock drift of $\frac{1}{2}\epsilon$ compared to s_3, whereas s_2 has a clock drift of $-\frac{1}{2}\epsilon$. The dashed arrow lines illustrate the temporal ordering of all events, as observed by each of the servers. From s_3's point of view, the resulting temporal sequence of events is $((s_3, t_1), (s_3, t_2), (s_2, t_3), (s_1, t_4), (s_2, t_5))$; from s_2's point of view, it is $((s_3, t_1), (s_2, t_3), (s_3, t_2), (s_2, t_5), (s_1, t_4))$; from s_1's point of view, it is $((s_3, t_1), (s_3, t_2), (s_2, t_3), (s_2, t_5), (s_1, t_4))$.

Our assumption that $t' \leq t_0 - \frac{1}{1-\rho}\epsilon$ and the clock monotony assumption (4.5) allow the following inference:

$$c_s(t') \leq c_s(t_0 - \frac{1}{1-\rho}\epsilon) \tag{4.12}$$

The assumption of a maximum clock drift rate (4.6) allows the following inference by substituting t by $t_0 - \frac{1}{1-\rho}\epsilon$ and δ by $\frac{1}{1-\rho}\epsilon$:

$$c_s(t_0 - \frac{1}{1-\rho}\epsilon) \leq c_s(t_0) - \epsilon \tag{4.13}$$

Putting together (4.11), (4.12) and (4.13) leads to the following inequality:

$$c_{s'}(t') \leq c_s(t_0 - \frac{1}{1-\rho}\epsilon) + \epsilon \leq c_s(t_0) \tag{4.14}$$

Consequently, $(s', t') \xrightarrow{l} (s, t_0) \vee (s', t') \parallel_l (s, t_0)$, subject to definitions 4.13 and 4.14, from which it follows that $(s', t') \in \mathcal{S}$, in accordance with definition 4.15 and our assumption that $\mathcal{S}$ is consistent on the supposition of loosely synchronized server clocks. □

Lemma 4.2. *A snapshot $\mathcal{S}$ with $\mathtt{cons}_l(\mathcal{S}, (s, t_0))$ does not contain any state-changing events (s', t') that occurred after the time $t_0 + \frac{1}{1-\rho}\epsilon$:*

$$\mathtt{cons}_l(\mathcal{S}, (s, t_0)) \wedge \exists (s', t') \in \mathcal{S}_{all}\ (t' > t_0 + \frac{1}{1-\rho}\epsilon) \Rightarrow (s', t') \notin \mathcal{S}$$

Proof. Let us assume that a state-changing event $(s', t') \in \mathcal{S}_{all}$ exists with $t' > t_0 + \frac{1}{1-\rho}\epsilon$. On s''s local clock, (s', t') occurred at $c_{s'}(t')$. The assumption of loosely synchronized server clocks (4.10) implies that:

$$c_{s'}(t') \geq c_s(t') - \epsilon \tag{4.15}$$

Our assumption that $t' > t_0 + \frac{1}{1-\rho}\epsilon$ and the clock monotony assumption (4.5) allow the following inference:

$$c_s(t') > c_s(t_0 + \frac{1}{1-\rho}\epsilon) \tag{4.16}$$

The assumption of a maximum clock drift rate (4.6) allows for the following inference by substituting t by t_0 and δ by $\frac{1}{1-\rho}\epsilon$:

$$c_s(t_0 + \frac{1}{1-\rho}\epsilon) \geq c_s(t_0) + \epsilon \tag{4.17}$$

Putting together (4.15), (4.16) and (4.17) leads to the following inequality:

$$c_{s'}(t') \geq c_s(t') - \epsilon > c_s(t_0 + \frac{1}{1-\rho}\epsilon) - \epsilon \geq c_s(t_0) \tag{4.18}$$

Consequently, $\neg((s',t') \xrightarrow{l} (s,t_0) \vee (s',t') \parallel_l (s,t_0))$, subject to definitions 4.13 and 4.14, from which it follows that $(s',t') \notin \mathcal{S}$, in accordance with definition 4.15 and our assumption that $\mathcal{S}$ is consistent on the supposition of loosely synchronized server clocks.

□

Theorem 4.1. *Let $\mathcal{S}_{s'} = \{(x,y) \in \mathcal{S} : x = s'\}$ be the state of a server s' in a snapshot $\mathcal{S}$. If $\mathtt{cons}_l(\mathcal{S},(s,t_0))$ is true, $\mathcal{S}_{s'}$ reflects the state of s' within a time frame $[t_0 - \frac{1}{1-\rho}\epsilon, t_0 + \frac{1}{1-\rho}\epsilon]$ i.e., there is a time t with $t_0 - \frac{1}{1-\rho}\epsilon \leq t \leq t_0 + \frac{1}{1-\rho}\epsilon$ and $\mathtt{state}_{s'}(t) = \mathcal{S}_{s'}$:*

$$\mathtt{cons}_l(\mathcal{S},(s,t_0)) \Rightarrow \forall s' \in S\ \exists t \in [t_0 - \frac{1}{1-\rho}\epsilon, t_0 + \frac{1}{1-\rho}\epsilon]\ (\mathtt{state}_{s'}(t) = \mathcal{S}_{s'})$$

Proof. Follows from definition 4.3 and lemmata 4.1 and 4.2. □

Theorem 4.2. *Let $\mathcal{S}_{s'} = \{(x,y) \in \mathcal{S} : x = s'\}$ be the state of a server s' in a snapshot $\mathcal{S}$. If $\mathtt{cons}_l(\mathcal{S},(s,t_0))$ is true, the states $\mathcal{S}_{s_1}$ and $\mathcal{S}_{s_2}$ of any two servers s_1 and s_2 can be expressed through server states $\mathtt{state}_{s_1}(t_1')$ and $\mathtt{state}_{s_2}(t_2')$ with $|t_2' - t_1'| \leq \frac{1}{1-\rho}\epsilon$:*

$$\mathtt{cons}_l(\mathcal{S},(s,t_0)) \Rightarrow$$
$$\forall s_1,s_2 \in S\ \exists t_1,t_2\ (\mathtt{state}_{s_1}(t_1) = \mathcal{S}_{s_1} \wedge \mathtt{state}_{s_2}(t_2) = \mathcal{S}_{s_2} \wedge |t_1 - t_2| \leq \frac{1}{1-\rho}\epsilon).$$

Proof. Let us assume that $\mathtt{cons}_l(\mathcal{S},(s,t_0))$ is true. According to the definitions 4.13, 4.14 and 4.15, this implies that any server state $\mathtt{state}_{s'}(t) \subseteq \mathcal{S}$ with $\mathtt{state}_{s'}(t) = \mathcal{S}_{s'}$ contains all those state-changing events (s',t') that fulfill the condition that $c_{s'}(t') \leq c_s(t_0)$. Consequently, there must be two points in time t_1, t_2 with $\mathtt{state}_{s_1}(t_1) = \mathcal{S}_{s_1}$ and $\mathtt{state}_{s_2}(t_2) = \mathcal{S}_{s_2}$ that fulfill the condition that $c_{s_1}(t_1) = c_{s_2}(t_2) = c_s(t_0)$.

Let us assume without loss of generality that $t_1 \geq t_2$ and $t_2 = t_1 - \delta$ i.e.,

$$c_{s_1}(t_1) = c_{s_2}(t_1 - \delta) \tag{4.19}$$

The assumption of a maximum clock drift rate (4.6) allows for the following inference by substituting t by $t_1 - \delta$:

$$c_{s_2}(t_1 - \delta) \leq c_{s_2}(t_1) - (1-\rho)\delta \tag{4.20}$$

The clock synchrony assumption (4.10) allows the following statement:

$$c_{s_2}(t_1) \leq c_{s_1}(t_1) + \epsilon \tag{4.21}$$

Putting together (4.19), (4.20) and (4.21) leads to the following inequality:

$$c_{s_1}(t_1) \leq c_{s_1}(t_1) + \epsilon - (1-\rho)\delta \tag{4.22}$$

Consequently, $(1-\rho)\delta \leq \epsilon$, which implies that $\delta \leq \frac{1}{1-\rho}\epsilon$. □

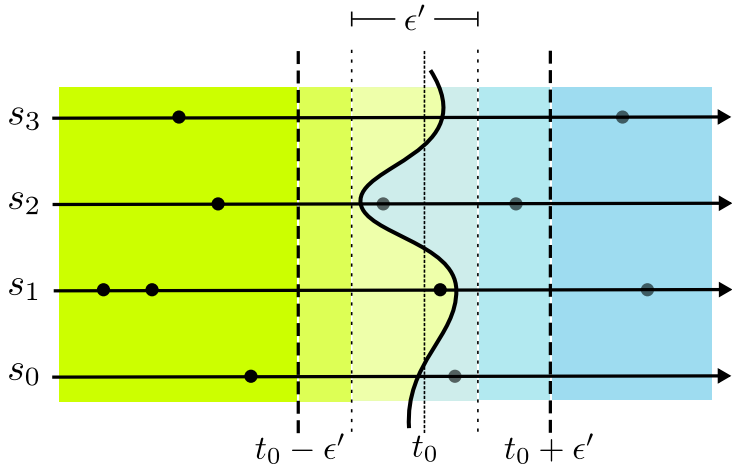

Figure 4.7: Illustration of snapshot semantics based on synchronized clocks

As stated in theorem 4.2, $\frac{1}{1-\rho}\epsilon$ sets an upper bound on the time span during which server states are effectively captured in a snapshot. According to theorem 4.1, this time span lies within the interval $[t_0 - \frac{1}{1-\rho}\epsilon, t_0 + \frac{1}{1-\rho}\epsilon]$. For a more compact presentation, we use the symbol ϵ' to denote the term $\frac{1}{1-\rho}\epsilon$. Because of $0 < \rho \ll 1$, ϵ can bee seen as a close approximation of ϵ'.

Figure 4.7 illustrates the principle of the "synchronized clocks" snapshot consistency model from a global observer's point of view. The serpentine line represents a cut, which is consistent according to definition 4.15. The behavior of this cut line is determined by the relative individual clock drift of each server. In particular, the points of intersection of the cut line with the time lines of all servers define a set of virtual events that happened concurrently according to definition 4.13. Although the cut line may look different if the momentary drift between individual server clocks changes, it is guaranteed to run within an ϵ' interval, which is covered by the interval $[t_0 - \epsilon', t_0 + \epsilon']$, as stated in theorems 4.1 and 4.2. It can thus be seen as an approximation of the vertical (dotted) cut line at t_0 that represents a point-in-time-consistent cut. The smaller ϵ and consequently ϵ' are, the more does the serpentine line converge to a vertical cut line. If ϵ is zero, $\texttt{cons}_l$ and $\texttt{cons}_g$ are equivalent.

4.4 Relationships between Consistency Models

Both causal consistency and consistency based on loosely synchronized clocks relax the concept of point-in-time consistency. In formal terms, this means that $\texttt{cons}_g$ implicates both $\texttt{cons}_c$ and $\texttt{cons}_l$: a point-in-time snapshot is both causally consistent and consistent on the supposition of loosely synchronized clocks. The former is true because $\texttt{cons}_g$ implies a global temporal ordering of all events, and causal precedence implies temporal precedence (which follows from (4.4) and definition 4.10), while the latter is true because $\texttt{cons}_g$ is a special case of $\texttt{cons}_l$ with $\epsilon = 0$. However, $\texttt{cons}_c$

neither implicates $\texttt{cons}_l$ nor does $\texttt{cons}_l$ implicate $\texttt{cons}_c$. A causally consistent snapshot is not necessarily consistent on the supposition of loosely synchronized clocks, as state-changing events may be captured within an unbounded time frame. A consistent snapshot on the supposition of loosely synchronized clocks may not be causally consistent, as the ordering of events solely depends upon server clock drift rather than communication between hosts.

4.4.1 Properties of Snapshots based on Synchronized Clocks

Enforcing causal consistency on a snapshot requires communication between hosts, as causal relationships among messages have to be tracked. In contrast, "synchronized clocks" consistency can be independently enforced by each server, without knowledge of the state of any other server. The consistency model qualifies for snapshots in large-scale distributed file system especially because it rules out communication as a potential bottleneck and source of error.

Anomalous Behavior. A total ordering of events based on physical clocks may lead to anomalous behavior, as pointed out by *Lamport* [Lam78]. In consequence, snapshots based on loosely synchronized clocks may not always reflect meaningful global states. A user who changes a file and requests a file system snapshot a moment later expects the change to be included in the snapshot. This cannot be guaranteed if the event of taking the snapshot may be regarded by the servers as having happened before the event of changing the file, in the wake of a minimal asynchrony between server clocks and despite the causal dependency between the two events. Similarly, a database system that compacts its data by first writing new index files and then truncating a log file containing the most recent updates may be captured in a state in which the updates are effectively lost if the event of truncating the log is regarded by the servers as having happened before the event of writing the new index files.

Fig. 4.8 depicts the problem of anomalous behavior. A client c_1 sends a message to a server s_2, which is received at time t_1. After having received s_2's response, it sends another message to server s_1, which is received at time t_2. Although (s_2, t_1) happened before (s_1, t_2) according to the causal dependency between these events, a snapshot that is consistent on the supposition of loosely synchronized clocks may include (s_1, t_2) but not (s_2, t_1) if $c_{s_2}(t_1) > c_{s_1}(t_2)$.

4.4.2 Restoring Causality

A possible way of resolving anomalous behavior and restoring causality in the "synchronized clocks" consistency model is to enforce a minimum message delay [Lam78] of more than ϵ' on a server:

$$\forall((h,t),(s,t')) \in M_{all} \cap ((H \times \mathbb{R}) \times (S \times \mathbb{R}))\ (t < t' - \epsilon') \tag{4.23}$$

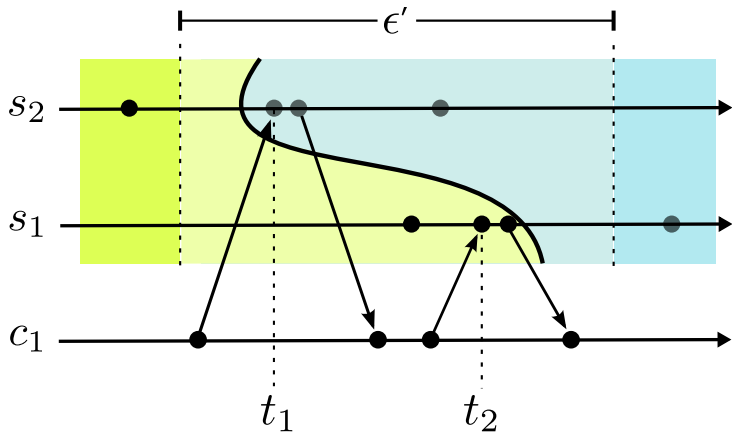

Figure 4.8: Violation of causal consistency with the "synchronized clocks" consistency model. The causal dependency between (s_2, t_1) and (s_1, t_2) ensures that (s_2, t_1) happened before (s_1, t_2); yet, (s_2, t_1) may be excluded from a snapshot while (s_1, t_2) is included if $c_{s_2}(t_1) > c_{s_1}(t_2)$.

Under these circumstances, consistency based on synchronized clocks implies causal consistency.

Theorem 4.3. *Assuming* (4.23), $\texttt{cons}_l(\mathcal{S}, (s, t_0))$ *implies* $\texttt{cons}_c(\mathcal{S}, (s, t_0))$.

Proof. We show that any snapshot $\mathcal{S}$ with $\texttt{cons}_l(\mathcal{S}, (s, t_0))$ satisfies $\texttt{cons}_c(\mathcal{S}, (s, t_0))$ under the terms of (4.23):

$$\mathcal{S} = \{(s', t') \in \mathcal{S}_{all} : \ (s', t') \xrightarrow{l} (s, t_0) \vee (s', t') \parallel_l (s, t_0)\} \Rightarrow$$
$$\mathcal{S} \supseteq \{(s', t') \in \mathcal{S}_{all} : \ (s', t') \xrightarrow{c} (s, t_0)\} \wedge \nexists (s', t') \in \mathcal{S} \ ((s, t_0) \xrightarrow{c} (s', t'))$$

More precisely, it is sufficient to show that:

$$(i) \qquad (s', t') \xrightarrow{c} (s, t_0) \Rightarrow (s', t') \xrightarrow{l} (s, t_0)$$
$$(ii) \qquad (s', t') \xrightarrow{l} (s, t_0) \vee (s', t') \parallel_l (s, t_0) \Rightarrow \neg((s, t_0) \xrightarrow{c} (s', t'))$$

If $s = s'$, (i) and (ii) are satisfied because '$\xrightarrow{c}$' and '$\xrightarrow{l}$' are equivalent to '$\rightarrow$' for events on the same server, according to definitions 4.10 and 4.13. For the case that $s \neq s'$, we start with a proof of (i):

Our assumption (4.23) along with definition 4.10 ensures that $t_0 - t' > \epsilon'$ if $(s', t') \xrightarrow{c} (s, t_0)$. According to definition 4.13, $(s', t') \xrightarrow{l} (s, t_0)$ is true iff $c_s(t_0) > c_{s'}(t')$. Thus, we need to show that $c_s(t_0) > c_{s'}(t')$ if $t_0 - t' > \epsilon'$.

The monotony assumption (4.5) and our assumption that $t_0 > t' + \epsilon'$ satisfy the following inequality:

$$c_s(t_0) > c_s(t' + \epsilon') \tag{4.24}$$

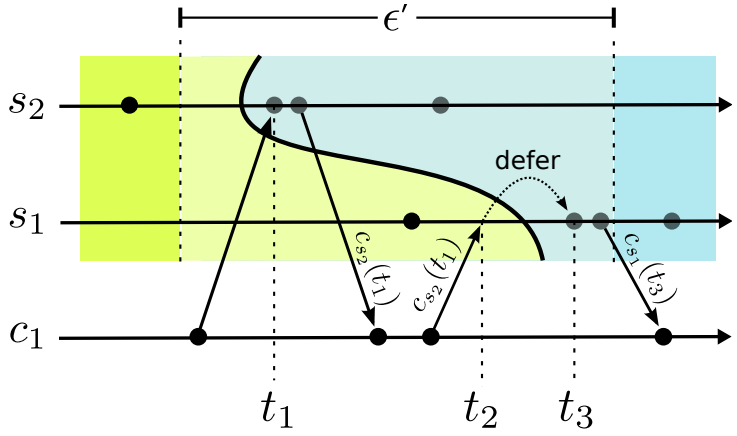

Figure 4.9: Preserving causality by means of server timestamps. The processing of (s_1, t_2) is deferred to t_3, which fulfills the condition that $c_{s_1}(t_3) > c_{s_2}(t_1)$.

From the definition of ϵ' and the clock drift assumption (4.6), it follows that:

$$c_s(t' + \epsilon') \geq c_s(t') + \epsilon \tag{4.25}$$

The clock synchrony assumption (4.10) implies the following:

$$c_s(t') + \epsilon \geq c_{s'}(t') \tag{4.26}$$

Putting together (4.24), (4.25) and (4.26) shows that $c_s(t_0) > c_{s'}(t)$.

We now show (ii). In accordance with definition 4.13 and 4.14, $(s', t') \xrightarrow{l} (s, t_0) \vee (s', t') \parallel_l (s, t_0)$ is true iff $c_{s'}(t') \leq c_s(t_0)$. Let us assume that an event (s', t') exists with $(s, t_0) \xrightarrow{c} (s', t')$. Because of (4.23), such an event must satisfy the condition that $t' > t_0 + \epsilon'$. Analogous to (4.24), (4.25) and (4.26), this implies that $c_{s'}(t') > c_s(t_0)$, which is in contradiction with our assumption.

□

To restore causal consistency in the aforementioned way, it is necessary to ensure that any message sent to a server is in transit for a time span of more than ϵ'. Estimates of ϵ and ϵ', accordingly, are typically in the range of tens of milliseconds on the Internet and milliseconds on local area networks [Mil95], which makes it likely that off-the-shelf network hardware satisfies the condition. However, we assume that the network does not give any guarantees on message transfer times. To ensure that the condition is always satisfied regardless of the underlying network, it is therefore necessary to artificially delay incoming messages on a server by ϵ' before processing them, which comes at the cost of an increased latency and reduced response time.

Moh et al. [ML04] sketched an alternative approach that preserves causality without necessarily suspending the processing of each request. They suggest to selectively de-

lay messages, such that any two state-changing events (s,t), (s',t') with $(s,t) \xrightarrow{c} (s',t')$ also satisfy $(s,t) \xrightarrow{l} (s',t')$. As shown in figure 4.9, this can be accomplished through timestamped server messages. The behavior can be best described with the following three rules:

- **Timestamping rule.** A server piggybacks its current clock time on any message sent to a client or remote server.
- **Forwarding rule.** A client forwards the largest received timestamp to any subsequently contacted server.
- **Deferral rule.** A server receiving a timestamped message defers its local processing until its current clock time is greater than the attached timestamp.

Thus, a server s' that receives a timestamp $c_s(t)$ enforces any state-changing event (s',t') with $(s,t) \xrightarrow{c} (s',t')$ to be delayed until $c_{s'}(t') > c_s(t)$, which by definition 4.13 ensures that $(s,t) \xrightarrow{l} (s',t')$. The approach only induces occasional delays in the processing of requests, which are typically very short. More specifically, clock synchrony ensures that these delays cannot exceed ϵ'.

4.5 Related Work: Distributed Snapshots

The problem of capturing consistent snapshots in distributed systems has been extensively studied in literature. Various solutions and algorithms have been proposed that enforce different consistency models on snapshots.

4.5.1 Causally Consistent Snapshots

The seminal paper on snapshots in distributed systems was published by *Chandy and Lamport* in 1985 [CL85]. It presents an algorithm for causally consistent snapshots, which later became popular as the *Chandy-Lamport Algorithm* for snapshots. The original motivation for the algorithm was the need to verify *stable properties* of a running system i.e., properties of the global system state that remain stable once they take effect. Examples of such properties are "the computation has terminated", "the system is deadlocked" or "the token passed between all processes was lost". To ensure that a causally consistent snapshot allows for the detection of stable properties, it is not sufficient to only observe the temporal sequence of events on each server. In addition, it is necessary to monitor the state of all communication channels in the system and incorporate possible in-transit messages into snapshots. Accordingly, *Kshemkalyani et al.* [KRS95] defined a snapshot as consistent if it fulfills the following two requirements:

(i) If the snapshot contains `send`(m), it must either contain m or `recv`(m).

(ii) If the snapshot does not contain `send`(m) it may neither contain m nor `recv`(m).

Hélary et al. [HPR90] address this aspect in a more precise manner by introducing the notion of *strong* and *weak* snapshot consistency. A snapshot that fulfills requirements (i) and (ii) offers *weak* consistency, as it reflects a global state in its entirety only if in-transit messages are taken into account. To offer *strong* consistency, a snapshot has to ensure that any sent message was also received i.e., it reflects any recorded *send* event together with the respective *receive* event:

(iii) The snapshot contains $\texttt{send}(m)$ if and only if it contains $\texttt{recv}(m)$.

Accordingly, a strongly consistent snapshot reflects a state that conforms to requirements (i) and (ii) without containing any in-transit messages.

Chandy-Lamport Algorithm. The Chandy-Lamport algorithm captures snapshots that fulfill the weak consistency requirements (i) and (ii), given that messages are delivered in FIFO order i.e., $\texttt{send}(m_1) \rightarrow \texttt{send}(m_2)$ implicates that $\texttt{recv}(m_1) \rightarrow \texttt{recv}(m_2)$ on any communication channel between two processes. In doing so, it resorts to specific marker messages, which separate those messages and events to be captured from those not to be captured.

Algorithm 4.1 sketches the Chandy-Lamport algorithm. To initiate a snapshot, a process acts according to the *Marker-Sending Rule*, which causes the process to send marker messages along all outgoing channels. Any process that receives a marker message acts according to the *Marker-Receiving Rule*. If such a process has not record its state yet, it records its state, spreads marker messages across all outgoing channels and starts recording all incoming channels except the one on which it received the marker. Any subsequently received marker message causes the process to stop its recording of the respective channel. The algorithm terminates as soon as all processes have received marker messages on all incoming channels. A formal presentation of the algorithm as well as a proof of correctness can be found in [CL85] and [Lyn96].

Optimizations and Enhancements. Since the appearance of Chandy's and Lamport's pioneering paper, a range of enhancements and optimizations to the algorithm have been published. *Venkatesan* [Ven89] presented an asymptotically message-optimal protocol that renders the repeated execution of the Chandy-Lamport algorithm more efficient by only sending marker messages on those channels that actually conveyed messages since the last snapshot was taken. *Spezialetti and Kearns* [SK86] optimized the concurrent recording of multiple snapshots, such that local states are only recorded and marker messages are only spread once per process, irrespective of the number of snapshots.

Hélary [H89] proposed an algorithm that generalizes the basic principle of the Chandy-Lamport algorithm. The algorithm employs a wave synchronization scheme to record a consistent global state, which visits each process exactly once e.g., by traversing a spanning tree or ring overlay. When a process is visited, it records its local state and spreads marker messages. A process receiving a marker message before having been visited delays all incoming messages on the respective channel until being visited.

Algorithm 4.1: Chandy-Lamport Algorithm (informal)

```
1  Marker-Sending Rule:
2      ○ record the process state
3      ○ begin recording the state of each incoming channel
4      ○ send a marker message on each outgoing channel
5
6  Marker-Receiving Rule:
7      if no state has been recorded yet then
8          ○ record the process state
9          ○ begin recording the state of each incoming channel, except the one on
10           which the marker was received
11         ○ send a marker message on each outgoing channel
12     else
13         ○ stop recording the state of the channel on which the marker was received
14     end
15
```

The scheme effectively separates the process of recording local states from the process of determining the set of events and messages to be recorded. The Chandy-Lamport algorithm can be seen as a special case of *Hélary*'s algorithm, in which a process is visited when receiving the first marker message.

In [HPR90], *Hélary et al.* describe how to extend the algorithm, such that it captures strongly consistent snapshots. To ensure that *send* and *receive* events of a message are recorded together in an atomic fashion, the algorithm keeps track of the number of messages sent and received on each channel. When a message is sent, a counter for the respective channel is incremented; when a message is received, it is decremented. Thus, a snapshot is guaranteed to offer strong consistency if the sum of all message counters is zero across all processes. To reach such a state, however, it may be necessary to trigger multiple wave sequences.

Algorithms for Non-FIFO Systems. Further related work addresses the problem of capturing snapshots in systems that do not offer FIFO message ordering guarantees. *Taylor* [Tay89] showed that algorithms recording consistent snapshots of such systems must either inhibit ongoing computations or piggyback control messages on computation messages.

A simple inhibitory approach is to implement a FIFO ordering of messages on top of all (non-FIFO) channels by delaying each outgoing message until the previously sent message has been acknowledged by its recipient. To enhance his algorithm for non-FIFO channels, *Hélary* [H89́] suggested to delay outgoing messages to any neighbor process until either an acknowledgment for the previous message or a marker message has been received from that process, or the local process has recorded its state.

A piggybacking algorithm was suggested by *Lai and Yang* [LY87], which is based on a coloring scheme for processes. A process can be colored *white* or *red*, depending on

whether it has already recorded its state. The color of a process is piggybacked on each message sent by the process. A process is initially colored white and becomes red when it receives a red message. When becoming red, it records its local state and spreads red marker messages along all channels. As a result, messages are white if and only if they were sent before the sender process recorded its local state. To ensure that white in-transit messages are included in a snapshot, each process keeps a history of all white messages sent and received along all channels. When sending a red marker message, the history of white messages sent on the respective channel is attached, which enables the receiving process to determine the set of in-transit messages on the channel by means of its local history of received messages.

A similar algorithm was proposed by *Li et al.* [LRV87]. To support repeated snapshots and concurrent snapshot initiations, they present a generalized tagging scheme for messages as an alternative to *Lai and Yang*'s white-and-red coloring scheme. Each invocation of the algorithm causes the initiator to tag messages with a globally unique identifier for the current snapshot, which makes it possible to record multiple snapshots independently of each other. *Li et al.* also obviate the need for message histories by only tracking and communicating numbers of messages sent and received along each channel. Such message counts provide the basis for calculating the number of in-transit messages between any two processes, which need to be received in order to terminate the process of taking a snapshot.

The principle of using message counters instead of message histories has also been adopted by *Mattern* [Mat93]. Each process maintains a vector of counters for all processes in the system. While the counter for the local process represents the total count of all white messages received along all channels as a negative number, counters for remote processes represent numbers of white messages sent along the respective channels. Similar to *Hélary*'s approach, snapshots are recorded through message waves that visit each process exactly once. A first wave colors each process red, captures the local state, and adds the local vector to a global vector representing the sum of all vectors. Thus, the resulting sum vector reflects the number of white messages that were still in-transit and have not yet been received by their target processes. A second wave waits on each process until all white messages have been received that were missing according to the sum vector and records these as part of the snapshot.

Scalability and Fault Tolerance. All of the aforementioned snapshot algorithms rely on communication to record snapshots. The number of messages required to capture a snapshot has a size of at least an $O(n)$, where n can be the number of processes, channels or concurrently initiated snapshots [KRS95]. The same applies to the message size, the internal state that needs to be kept, as well as the response time of the algorithm.

To mitigate scalability problems, *Garg et al.* [GGS06, GGS10] and *Kshemkalyani* [Ksh10] proposed different algorithms that allow a trade-off between message size, message complexity, space complexity and response time and thus ensure a sublinear growth for a subset of these items. These algorithms internally resort to different overlays on top of the network topology, which ensure that control messages are only propagated

Algorithm	Message Complexity	Space Complexity	Response Time
Chandy-Lamport [CL85]	$O(n^2), O(e)$	$O(1)$	$O(e)$
Venkatesan [Ven89]	$O(n^2), \Omega(n+u)$	$O(1)$	$O(u)$
Spezialetti and Kearns [SK86]	$O(rn^2), O(e)$	$O(r)$	$O(e)$
Hélary [Hél89, HPR90]	$O(e)$	$O(1)$	$O(e)$
Lai and Yang [LY87]	$O(n)$	unbounded	$O(n)$
Li et al. [LRV87]	$O(n)$	$O(1)$	$O(n)$
Mattern [Mat93]	$O(n^2)$	$O(n^2)$	$O(n)$
Garg et al.: Grid-Based [GGS06, GGS10]	$O(n\sqrt{n})$	$O(n^2)$	$O(\sqrt{n})$
Garg et al.: Tree-Based [GGS06, GGS10]	$O(n\ log(n)\ log(m))$	$O(n\ log(n)\ log(m))$	$O(n\ log(n)\ log(m))$
Garg et al.: Centralized [GGS06, GGS10]	$O(n\ log(m))$	$O(n\ log(m))$	$O(n\ log(m))$
Kshemkalyani: Simple Tree [Ksh10]	$O(n)$	$O(n^2)$	$O(log(n))$
Kshemkalyani: Hypercube [Ksh10]	$O(n\ log(n))$	$O(n^2)$	$O(log(n))$

Table 4.2: Scalability of snapshot algorithms, based on [KRS95], [Ksh10], [GGS06]. *Message complexity* reflects the total number of messages to capture a snapshot; *space complexity* reflects the consumed capacity in terms of message volumes and required local memory; *response time* reflects the latency between initiation and termination. Furthermore, n = number of processes, e = number of channels, u = number of channels on which messages were sent since last snapshot, r = number of concurrently initiated snapshots, m = average number of in-transit messages per process.

along a subset of all channels. A comparison of snapshot algorithms with respect to their scalability is shown in table 4.2.

Another issue is failure handling and fault tolerance. The mean time to failure in the overall system decreases exponentially with the number of nodes in the system [AGGM04], which can affect the liveness of a snapshot algorithm. Most of the aforementioned algorithms may fail to terminate if processes or channels fail to handle or transmit messages; inhibitory schemes can even cause ongoing computations to block infinitely.

As a solution, *Shah and Toueg* [ST84] published an enhanced version of the Chandy-Lamport algorithm that captures consistent snapshots in the face of message loss. By detecting lost messages through timeouts, the algorithm excludes failed processes and restricts consistency to those processes that responded within a well-defined time span.

4.5.2 Point-in-Time-Consistent Snapshots

Point-in-time consistency is a common property of snapshots in local file systems. If state is not physically distributed, such a consistency model is fairly easy to ensure, as all changes to files and directories can be totally ordered by the times at which they were executed. Most local file systems tag modifications with timestamps, which may either be bound to real time [BM07, HLM94, Qui91] or logical and incremented when a snapshot is taken [PB05a].

Enforcing point-in-time consistency on snapshots of a distributed file system makes it necessary to inhibit changes while taking the snapshot, such that the system state can be recorded without being exposed to changes. *Azagury et al.* [AFSM02] sketched an inhibitory scheme for snapshots in object-based file systems. A snapshot process first locks the entire set of files, marks all objects pertaining to these files to be copied on any subsequent write, and finally releases the locks.

However, such a scheme limits the scale and fault tolerance of the system installation. The duration of a snapshot operation depends upon the number of hosts, files, as well as message transmission times, which may render the system unavailable for a significant time if large numbers of files on globally distributed hosts are involved. Moreover, any host failure during the process of taking a snapshot will cause the entire system to become unavailable for an unbounded period of time.

4.5.3 Clock-Based Snapshot Consistency

Liskov [Lis93] suggested that loosely synchronized clocks can help to reduce communication in distributed systems and thus improve the performance of distributed algorithms. *Moh and Liskov* [ML04] picked up on this idea and presented a non-inhibitory, time-based snapshot scheme for TimeLine, an archive service for a distributed storage system. The system timestamps all modifications and captures all those changes to data objects that appear to have occurred before the snapshot was requested, based on the principle of "synchronized clocks" consistency. To prevent causality violations

and anomalous behavior, messages are timestamped and executions are delayed in the same way as described in section 4.4.2.

Several checkpointing algorithms for distributed applications resort to loosely synchronized clocks in order to record snapshots across multiple processes without communication. Under the assumption of an approximated global time, a process that initiates a checkpoint can be sure that the checkpointing procedure has terminated after a time span that equals the sum of the maximum clock drift and the minimum time to detect a failure [EAWJ02, Tre05].

To ensure (causal) consistency of such a snapshot, *Tong et al.* [TKT92] suggest to record channel states by keeping a record of sent messages until their delivery has been acknowledged by their recipients, and by adding a checkpoint number to each message and acknowledgment. A similar approach is followed by *Cristian and Jahanian* [CJ91]. *Ramanathan and Shin* [RS93] presented a time-based protocol that utilizes clock synchrony to reduce blocking times and the number of checkpoints that need to be retained.

As an alternative to the approach of maintaining synchronized clocks along with message records and a piggybacking scheme for checkpoint numbers, *Neves and Fuchs* [NF96] proposed an algorithm based on relative timers. A checkpoint is initiated by starting timers across all processes and recorded when a timer expires. Consistency is ensured by inhibiting communication; when a timer expires, no messages are sent for a period of time that depends upon an estimated maximum message transfer time and clock drift rate.

4.6 Summary and Results

We presented and discussed three different consistency models for snapshots:

- *point-in-time consistency*, which requires server states to be captured at the exact same point in time;
- *causal consistency*, which requires server states to be consistent with the ordering of events stipulated by the sending and delivery of messages;
- *"synchronized clocks" consistency*, which requires server states to be captured within a well-defined, narrow time frame.

Table 4.3 shows a comparison of the different consistency models. While point-in-time consistency can only be enforced with significant restrictions on scalability, availability and fault tolerance, consistency based on loosely synchronized clocks can be enforced without any such restrictions. To define an intuitive and meaningful semantics for file system snapshots, however, it is necessary to take account of the order of changes to files and directories that is prescribed by each client. Only causally consistent snapshots guarantee to reflect global states that incorporate all changes in the correct order, which is necessary to ensure that a snapshot reflects sets of files in a usable state from an application's point of view.

	Point-in-Time Consistency	**Causal Consistency**	**"Synchronized Clocks" Consistency**
timing model	real time	virtual time	real time
causal ordering of changes	yes	yes	only with enhancements
scalability	none	limited	unlimited
availability	read-only during snapshots	mostly no restrictions; some algorithms may inhibit changes	no restrictions; occasional minimal delays with causality enhancements
failure impact	recording fails to terminate	most algorithms may fail to terminate	none
response time (n = #servers)	$O(n)$	typically $\geq O(n)$	$O(1)$

Table 4.3: Comparison of snapshot consistency models based on their properties

There is an extensive body of previous work on causally consistent snapshots in distributed systems. Most published algorithms rely on specific control messages in order to record a snapshot. This can be a limiting factor for the scalability of a distributed file system, as increasing the number of hosts and network links also increases the number of control messages. Furthermore, most of these algorithms do not provide sufficient failure tolerance to guarantee progress in the face of host crashes and network link outages, which, however, are frequent in large-scale distributed systems.

In consideration of these facts and the initially stated requirements in terms of scalability, non-disruptiveness and crash-resilience, we come to the conclusion that the consistency model based on synchronized clocks with superimposed mechanisms to preserve causality offers the best alternative for a large-scale distributed file system. It scales to any number of hosts, as it does not require the dissemination of any control messages. It terminates after a well-defined time span of ϵ', regardless of the size of the system. It also comes with an immanent failure tolerance, as failed hosts will implicitly be excluded from a snapshot without affecting consistency or termination of the algorithm.

5 A Snapshot Algorithm for Object-Based File Systems

The previous chapters gave a comprehensive overview of versioning techniques and snapshot consistency models, which provide the basis for snapshots in large-scale distributed file systems. This chapter presents the design of a distributed snapshot algorithm for object-based file systems. The algorithm implements consistency based on loosely synchronized clocks and preserves causality through timestamped server messages, thus guaranteeing high scalability and restricting the impact of failures to the components on which they occur.

The chapter starts with a summary of assumptions on the object-based file system model along with a model of the internal interface (5.1). The following sections formally describe a version management scheme for file content (5.2) and metadata (5.3), as well as their functional interaction to provide for a snapshot scheme (5.4), followed by a description of enhancements that are necessary to ensure a POSIX compliant behavior in the face of truncated files and sparse files (5.5).

5.1 Modeling Object-Based File Systems

In accordance with the most common properties of object-based file systems described in section 2.3, we assume that file content and metadata is independently managed by different server types, viz object storage devices (OSDs) and metadata servers (MDS's). A client exposes a file system interface to users and applications and interacts with the servers in response to file system calls. Interactions are based on a synchronous request-reply pattern, which can, e.g., be implemented with synchronous remote procedure calls. All communication is initiated by the client. When receiving a request, a server processes the request in an atomic fashion and sends a response back to the client.

5.1.1 State and Interface

Each MDS persistently stores the metadata of its local volumes. Each OSD holds a persistent set O_f of objects for each of its files f. We formally define an object $(o, d) \in O_f$ as a tuple consisting of an object identifier o and a data item d representing the object's data on the local storage device.

Algorithm 5.1 shows the internal interface of an object-based file system, along with all relevant operations for versioning and snapshots. To reduce the complexity of our model, we restrict all accesses to entire objects rather than byte ranges within objects. Besides, we omitted any security-related processing steps.

Algorithm 5.1: Operations in an object-based file system

Client

```
    resolves the given path path to a file
1   function open(path)
2       send OPEN(path) to MDS                          open file on MDS
3       receive FILE(f) from MDS                        wait for response
4       return f                                        return file identifier

    reads an object o of a file f and returns the data
5   function read(f, o)
6       send READ_OBJECT(f, o) to OSD                   read object on OSD
7       receive DATA(d) from OSD                        wait for response
8       return d                                        return data

    writes an object o of a file f with the given data d
9   procedure write(f, o, d)
10      send WRITE_OBJECT(f, o, d) to OSD               write object on OSD
11      receive ACK() from OSD                          wait for acknowledgment

    closes a file f
12  procedure close(f)
13      send CLOSE(f) to OSD                            close file on OSD
14      receive ACK() from OSD                          wait for acknowledgment
```

MDS

```
    retrieves and returns the file for the given path path
15  on OPEN(path)
16      f ← file for path path                          retrieve file
17      send FILE(f) to Client                          send back file
```

OSD

```
    writes the object o of a file f with the data d
18  on WRITE_OBJECT(f, o, d)
19      O_f ← (O_f \ {(o', d') ∈ O_f : o' = o}) ∪ {(o, d)}   update object
20      send ACK() to Client                            send acknowledgment

    reads the object o of a file f and returns the corresponding data item
21  on READ_OBJECT(f, o)
22      d ← d' : (o, d') ∈ O_f                          retrieve object
23      send DATA(d) to Client                          send back data

    closes the file f
24  on CLOSE(f)
25      discard any open state
26      send ACK() to Client                            send acknowledgment
```

5.1.2 Request Times and Server Clocks

In accordance with the timed asynchronous system model [CF99], we assume that all requests are timed. t_{recv} denotes the point in time at which the currently processed request has been received. We assume that each server has a local clock c, where $c(t)$ denotes the current local clock time at the point in time t. We further assume that all server clocks are loosely synchronized as required by (4.10).

5.2 File Content Versioning

To allow for a versioning of file content, we assume that OSDs record versions of objects and files. Accordingly, object versions are augmented with a version identifier v, such that each object is attached to an individual version $(o, v, d) \in O_f$. To distinguish between older and newer versions, we further assume that a total ordering '$\leq$' is defined on version identifiers, where $v \leq v'$ denotes that the version bound to v is older than or equal to the one bound to v'.

5.2.1 Object Versioning

At its core, an OSD modifies objects of a file in response to write operations. A fundamental step in doing this is to persistently apply changes to existing objects. This can either be done by directly modifying the data item of an existing object, or by persistently recording the change as a new version and thus preserving the prior version against being overwritten.

Algorithm 5.2: Modifying and retrieving versioned objects

```
   updates the newest version of the object o of a file f with the new version v_new and data d_new
1  procedure updateObjectVersion(f, o, v_new, d_new)
2      (o, v_old, d_old) ← getObjectVersion(f, o, v_new)        retrieve newest object version
3      O_f ← (O_f \ {(o, v_old, d_old)}) ∪ {(o, v_new, d_new)}   update newest object version

   creates a new version of the object o of a file f with the version v_new and data d_new
4  procedure addObjectVersion(f, o, v_new, d_new)
5      O_f ← O_f ∪ {(o, v_new, d_new)}                           add new object version

   returns the newest version of an object o of a file f that is not newer than v
6  function getObjectVersion(f, o, v)
7      V ← {(o', v', d') ∈ O_f : o' = o ∧ v' ≤ v ∧               get newest object version ...
8           ∄(o, v'', d'') ∈ O_f (v' < v'' ≤ v)}                 ... not newer than v
9      if V = ∅ then                                             if no such version exists ...
10         return (o', ⊥, ⊥)                                     ... return empty version
11     else                                                      if version exists ...
12         return (o', v', d') ∈ V                               ... return version
```

Algorithm 5.2 provides a formal presentation of the primitives for the manipulation and retrieval of objects in a versioning-aware OSD. Object versions can either be updated, which effectively discards the previous version (lines 1-3), or added, which creates a new version while keeping the previous one (lines 4-5). They can be retrieved by means of version identifiers. Since each version remains up to date until a newer version is added, an object version lookup with an identifier v returns the newest version with an identifier $v' \leq v$ (lines 7-8). If no such version exists, $\perp$ is used to denote a missing version identifier or data item, with $\forall v \neq \perp \ (v > \perp)$ (line 10).

5.2.2 File Versioning

The two primitives for adding and updating object versions constitute the building blocks for the retention of file versions. A file version comprises a specific version of each object and thus defines a certain immutable state of the file's content.

Assuming that version identifiers assigned to object and file versions are strictly monotonically increasing like timestamps generated by a local clock, a single version identifier is sufficient to clearly define a file version. In addition to its set of versioned objects, each OSD also persistently keeps a record of version identifiers V_f for each of its files f. As shown in algorithm 5.3, it is sufficient to add a new version identifier to V_f in order to record a new file version (lines 1-2). Retrieving a file version works in a similar way as retrieving an object version (lines 3-8).

Algorithm 5.3: File versioning and version retrieval

```
   creates a new version v of the file f
1  procedure addFileVersion(f, v)
2      V_f ← V_f ∪ {v}                                   add file version

   returns the newest version of a file f that is not newer than v
3  function getFileVersion(f, v)
4      V ← {v' ∈ V_f : v' ≤ v ∧ ∄v'' ∈ V_f (v' < v'' ≤ v)}   get newest version not newer than v
5      if V = ∅ then                                     if no such version exists ...
6          return ⊥                                      ... return empty version
7      else                                              if version exists ...
8          return v' ∈ V                                 ... return version
```

Depending on the requirements presented by users and applications, file versions can be either created "on every write" or "on close". We decided for these two schemes among those described in section 3.1.4, as they are the most common ones and particularly easy to formalize and implement.

Version "On Every Write". Algorithm 5.4 records a file version "on every write" i.e., each time an object is written. After receiving a `write` request for an object, it may be necessary to wait until the local clock has advanced to a time later than the piggybacked

Algorithm 5.4: File versioning: "version on every write"

```
1 on WRITE_OBJECT(f, o, d, ts)
2     wait t_w, such that c(t_recv + t_w) ≥ ts      wait to preserve causality if necessary
3     ts_now ← c(t_recv + t_w)                      get timestamp as version identifier
4     addObjectVersion(f, o, ts_now, d)             add new timestamped object version
5     addFileVersion(f, ts_now)                     add new timestamped file version
6     send ACK(ts_now) to Client                    piggyback timestamp on response
```

timestamp ts (line 2), so as to avoid anomalous behavior as described in section 4.4.2. If server clocks are closely synchronized, however, it is likely that $c(t_{recv}) > ts$ because of previous message delays, which effectively means that no waiting is necessary. Subsequently, a new object version is added and a new file version is recorded; both receive a timestamp ts_{now} from the OSD's local clock as a version identifier (lines 3-5). Finally, the current clock time is sent back to the client in order to be piggybacked on follow-up requests (line 6).

Version "On Close". Since a "version on every write" scheme is often considered impractical because of its considerable demand in terms of storage and I/O capacity [MRH09], a common alternative is to create file versions when files are closed after writing [McC90, MRWHZ04, SFHV99, SGSG03]. Effectively, this means that new object versions need to be created only if the respective object has not been written yet since the file was opened. To keep track of those objects that have already been written during the current "open-close" period, each OSD holds a transient set W_f for each open file f. Depending on whether an object to be written is already contained in W_f, the newest object version is either updated, or a new object version is added.

Algorithm 5.5 illustrates the principle. It resembles algorithm 5.4, except that new object versions are only created once per open-close period, and new file versions are recorded only when files are closed after writing. Steps for preserving causality i.e., the initial waiting and the piggybacking of the current server time on the response, are deferred to the time when files are closed.

5.3 Metadata Versioning

Typically, metadata servers only make up a relatively small portion of all servers. To avoid bottlenecks in the metadata access path, it is of major importance that the overhead caused by recording metadata versions is kept as low as possible. We therefore assume that an MDS is capable of recording metadata versions by capturing point-in-time snapshots of a volume's complete metadata with a minimal impact on performance and availability.

Each metadata version defines the metadata of a file system snapshot. Algorithm 5.6 describes the management of metadata versions. To keep track of recorded metadata

Algorithm 5.5: File versioning: "version on close"

```
1  on WRITE_OBJECT(f, o, d)
2      ts_now ← c(t_recv)                          get timestamp as version identifier
3      if o ∈ W_f then                             if object has already been written ...
4          updateObjectVersion(f, o, ts_now, d)    ... update object version
5      else                                        if object has not been written yet ...
6          addObjectVersion(f, o, ts_now, d)       ... add new object version
7          W_f ← W_f ∪ {o}                         ... mark object as written
8      send ACK() to Client                        send acknowledgment

9  on CLOSE(f, ts)
10     wait t_w, such that c(t_recv + t_w) ≥ ts    wait to preserve causality if necessary
11     ts_now ← c(t_recv + t_w)                    get timestamp as version identifier
12     addFileVersion(f, ts_now)                   add new timestamped file version
13     W_f ← ∅                                     clear set of written objects
14     send ACK(ts_now) to Client                  piggyback timestamp on response
```

Algorithm 5.6: Management of metadata versions

Client

```
   triggers the recording of a new file system snapshot
1  procedure snapshot()
2      send SNAPSHOT() to MDS                      trigger snapshot creation
3      receive ACK(ts) from MDS                    wait for acknowledgment
```

MDS

```
   records a new metadata version
4  on SNAPSHOT()
5      M ← new metadata version                    capture new metadata version
6      wait t_w > ε'                               wait at least ε' to ensure stability and preserve causality
7      ts_now ← c(t_recv + t_w)                    get timestamp as version identifier
8      V ← V ∪ {(ts_now, M)}                       record timestamped metadata version
9      send ACK(ts_now) to Client                  piggyback timestamp on response

   returns the newest metadata version that is not newer than v
10 function getMetadataVersion(v)
11     V' ← {(v', M') ∈ V : v' ≤ v ∧               retrieve newest metadata version ...
12           ∄(v'', M'') ∈ V (v' < v'' ≤ v)}       ... not newer than v
13     if V' = ∅ then                              if no such version exists ...
14         return (⊥, ⊥)                           ... return empty version
15     else                                        if version exists ...
16         return (v', M') ∈ V'                    ... return version
```

versions, each MDS persistently stores a set $V \subset \{(v, M)\}$, where v reflects a version identifier and M a data structure representing the respective metadata. In response to a snapshot request sent by a client, a new metadata version is recorded in V with a locally assigned timestamp (lines 5-8). Metadata versions are retrieved in a similar manner as object and file versions (lines 10-16).

Apart from preserving causality, it is necessary to ensure that a snapshot remains immutable once it becomes accessible. Thus, insertions in V are delayed by some time t_w greater than ϵ' and timestamped with $c(t_{recv} + t_w)$ (lines 6-7). Along with the synchrony of all server clocks, this artificial delay ensures causality as described in section 4.4.2, and prevents the snapshot from being accessed while changes may still be included during the time interval $[t_{recv}, t_{recv} + \epsilon']$.

5.4 Accessing Snapshots

Algorithm 5.7 outlines the functions and message handlers on the client and server side that are necessary to read data from a file in a snapshot. Accessing files in a snapshot takes place in two steps: first, the client contacts the MDS to resolve the path name and retrieve the snapshot timestamp (lines 1-4); then, it contacts the OSD to read data from the file (lines 5-8), which in turn has to retrieve the respective file and object version.

A client attaches an access timestamp ts_a to its initial `open` request to the MDS (line 2). The access timestamp defines an upper bound for the timestamp ts_s of the metadata version. More precisely, the MDS selects the most recent metadata version (ts_s, V) in M with a timestamp less or equal to ts_a and resolves the path on it (lines 10-11).

Reading from a file in a snapshot requires the correct file content version to be retrieved, subject to definition 4.15. The client therefore attaches ts_s to any `read` request sent to an OSD (line 6). An OSD that receives such a timestamp selects the latest file version v_f in V_f that is not newer than ts_s (line 14). Having retrieved v_f, it looks up the latest version of the requested object that is not newer than v_f and sends back the attached data (lines 15-16).

5.5 Versioning of Sparse and Truncated Files

The POSIX `truncate` operation allows to explicitly set the length of a file, which can cause data to be cut off and discarded or implicitly appended. Furthermore, POSIX allows to seek to an offset beyond the current length of the file and perform a `write` operation, which implicitly adds missing chunks of data in the middle of the file. Files with such implicitly added, missing chunks of data are generally referred to as *sparse files*. POSIX stipulates that missing byte ranges of a sparse file are read as binary zeros, whereas attempts to read a file beyond its length have to result in truncated chunks of data indicating the end of the file [IEE08].

A file version needs to be aware of its length, so as to be able to determine if a `read` operation is performed beyond the end of the file. Since the truncate operation violates

Algorithm 5.7: Accessing snapshots

Client

```
    resolves the given path path to a file in the snapshot identified through ts_a
1   function open(path, ts_a)
2       send OPEN(path, ts_a) to MDS                    open file on MDS
3       receive FILE(f, ts_s) from MDS                  wait for response
4       return (f, ts_s)                                return file with snapshot timestamp

    reads an object o of a file f in a snapshot identified through the timestamp ts_s
5   function read(f, o, ts_s)
6       send READ_OBJECT(f, o, ts_s) to OSD             read object on OSD
7       receive DATA(d) from OSD                        wait for response
8       return d                                        return data
```

MDS

```
    retrieves and returns the file for the given path path in the snapshot identified through ts_a
9   on OPEN(path, ts_a)
10      (ts_s, M_s) ← getMetadataVersion(ts_a)          get metadata version
11      f ← file for path path in M_s                   resolve path on metadata snapshot
12      send FILE(f, ts_s) to Client                    send back resolved file and timestamp
```

OSD

```
    reads the object o of the newest version of a file f that is not newer than v and returns the data item
13  on READ_OBJECT(f, o, v)
14      v_f ← getFileVersion(f, v)                      retrieve file version v_f
15      (o, v_o, d) ← getObjectVersion(f, o, v_f)       retrieve object version
16      send DATA(d) to Client                          send back data
```

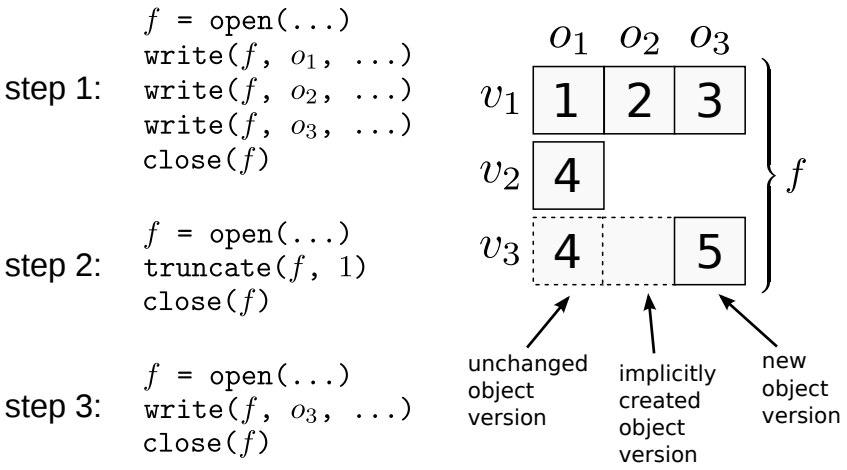

Figure 5.1: Versioning of truncated and sparse files. The left-hand side shows the steps that are necessary to create the file versions v_1, v_2, v_3 depicted on the right-hand side. For simplicity, version identifiers are denoted by the numbers 1 to 5 instead of physical timestamps.

the principle of nondecreasing file lengths across versions, it is not sufficient to determine the length of a version through the set of latest object versions with identifiers smaller than the file version identifier. The same is true if missing objects in a sparse file version are read.

The problem is illustrated in figure 5.1. A file with 3 objects is initially created (step 1), which is then truncated to the size of a single object (step 2). Finally, the third object is written, which effectively restores the initial file size of three objects but leaves a gap for the second object (step 3). Accordingly, v_2 should be bound to the set of object versions $\{(o_1, 4, ...)\}$ but algorithm 5.7 binds it to $\{(o_1, 4, ...), (o_2, 2, ...), (o_3, 3, ...)\}$. Similarly, v_3 should be bound to $\{(o_1, 4, ...), (o_3, 5, ...)\}$ instead of $\{(o_1, 4, ...), (o_2, 2, ...), (o_3, 5, ...)\}$ and return a zero-filled buffer when reading o_2.

A simple workaround is to create specific "gap" and "end-of-file" object versions as markers for deleted or zero-filled objects. However, this is not practical, as single invocations of `write` or `truncate` might cause numerous such versions to be created and thus involve a considerable overhead in truncating files and creating sparse files.

5.5.1 Length-Aware File Versioning

To ensure that versioning incorporates truncated file versions and sparse files in an efficient manner, OSDs have to provide additional mechanisms for the management of file and object versions. In particular, it is necessary to keep track of the length of all file versions. Accordingly, l_f denotes the length of the file, and V_f contains tuples (v, l) rather than only version identifiers v, where l represents the length of the file version.

Algorithms 5.8 and 5.9 show the respective changes that are necessary when creating and accessing file versions. Truncating files (lines 4-6) as well as adding new object

Algorithm 5.8: Versioning of sparse and truncated files

Client

```
   truncates a file f to the length l
1  procedure truncate(f, l)
2      send TRUNCATE(f, l) to OSD                    truncate file on OSD
3      receive ACK() from OSD                        wait for acknowledgment
```

OSD

```
   truncates a file f to the length l
4  on TRUNCATE(f, l)
5      l_f ← l                                       set current length to l
6      send ACK() to Client                          send acknowledgment

   creates a new version of the object o of a file f with the version v_new and data d_new
7  procedure addObjectVersion(f, o, v_new, d_new)
8      O_f ← O_f ∪ {(o, v_new, d_new)}               add new object version
9      l_f ← length of f                             update current length (if changed)

   creates a new version v of the file f
10 procedure addFileVersion(f, v)
11     V_f ← V_f ∪ {(v, l_f)}                        record version identifier v and length l_f

   returns the newest version of a file f that is not newer than v
12 function getFileVersion(f, v)
13     V ← {(v', l') ∈ V_f : v' ≤ v ∧                get newest version ...
14          ∄(v'', l'') ∈ V_f (v' < v'' ≤ v)}        ... not newer than v
15     if V = ∅ then                                 if no such version exists ...
16         return ⊥                                  ... return empty version
17     else                                          if version exists ...
18         return (v', l') ∈ V                       ... return version
```

Algorithm 5.9: Versioning of sparse and truncated files (continued)

```
    returns the newest version of a file f before v
19  function getFileVersionBefore(f, v)
20      V ← {(v′, l′) ∈ V_f : v′ < v ∧                    get newest version ...
21           ∄(v″, l″) ∈ V_f (v′ < v″ < v)}                ... before v
22      if V = ∅ then                                      if no such version exists ...
23          return ⊥                                       ... return empty version
24      else                                               if version exists ...
25          return (v′, l′) ∈ V                            ... return version

    returns the newest version of an object o of a file f with length l that is not newer than v
26  function getObjectVersion(f, o, v, l)
27      (o, v_o, d) ← getObjectVersion(f, o, v)            retrieve object version
28      l′ ← l
29      while v ≥ v_o do                                   check if object is missing
30          if o is beyond l′ then                         if object is either gap or beyond EOF ...
31              if o is beyond l then                      if object is contained in v ...
32                  return zero-length object              ... read object beyond EOF
33              else                                       if object is beyond EOF in v ...
34                  return zero-padded object              ... read gap in sparse file
35          (v, l′) ← getFileVersionBefore(f, v)           get next earlier file version
36      return (o, v_o, d)                                 return found version if not gap or EOF

    reads the object o of the newest version of a file f that is not newer than v and returns the data item
37  on READ_OBJECT(f, o, v)
38      (v_f, l) ← getFileVersion(f, v)                    retrieve file version
39      (o, v_o, d) ← getObjectVersion(f, o, v_f, l)       retrieve object version
40      send DATA(d) to Client                             send back data
```

versions (lines 7-9) updates the current length of the file, which is recorded with each new file version (lines 10-11). When retrieving a specific object version through a `read` request on a snapshot, it is necessary to check if the object version is bound to any older file version than the one in the snapshot. More specifically, all those file versions need to be checked that were created between the points in time of recording the object version and the requested file version (lines 29-35). If the length of any of these file versions is too small to include the object, the result is either a zero-length object representing the end of file (line 32), or a zero-padded object representing a gap in a sparse file (line 34), depending on whether the object is contained in the file version bound to the snapshot.

5.5.2 Interleaved Writes and Truncates

The algorithm presented above restricts the number of truncations to at most one per open-close period. More precisely, files that have been truncated may not be truncated or written anymore before being closed, since changing the file size multiple times without recording file versions could violate the aforementioned POSIX semantics of `truncate` operations and sparse files.

To ensure that files can be written and truncated repeatedly before being closed, it is necessary that the OSD keeps an internal record of the chronology of local `write` and `truncate` operations. For `write` operations, this record exists already in the form of timestamped file and object versions. Accordingly, all `write`s and `truncate`s can be totally ordered by recording timestamped versions of the current file length along with each `truncate` operation. With a "version on every write" policy, it is obvious that this can be done by adding a new file version when performing a `truncate` operation. With a "version on close" policy, file versions are not persistently recorded when files are truncated; instead, they are recorded in a transient *truncate record*, which is part of the file's open state. When the file is closed, a file version is persistently added for each truncate record entry before the record is discarded and the final file version is added.

6 Implementation and Practical Challenges

Based on the formal description of the snapshot algorithm for object-based file systems presented in the previous chapter, this chapter addresses the implementation of the algorithm along with its practical aspects and challenges.

We implemented the algorithm as part of XtreemFS [HCK+07], a scalable object-based file system. The implementation employs the versioning scheme described in chapter 3 in order to record file content and metadata versions. It guarantees snapshot consistency through timestamps from loosely synchronized server clocks and time-stamped server messages, as described in chapter 4.

The chapter starts with an overview of the implementation (6.1) and addresses practical challenges, which involve the deletion of snapshots and the removal of redundant versions (6.2), the execution of atomic operations (6.3), as well as the integration with replication (6.4).

6.1 Overview

In accordance with the algorithm presented in chapter 5, a consistent snapshot of a volume can be captured solely by recording new a metadata version on the MRC. File and object versions are independently managed by their OSDs and have their own life cycles. They are created when files are written, as described in section 3.3. File versions are loosely coupled with their metadata versions by means of their timestamps; appropriate file and object versions are retrieved when files are accessed on a snapshot, as described in section 5.4.

6.1.1 Capturing Snapshots

XtreemFS comes with a tool that allows users to create, list and delete snapshots. The tool is capable of capturing snapshots of entire volumes, individual directory subtrees, or single directories without any subdirectories. To restrict the selection of directories to be included, the MRC causes BabuDB to apply a filter to its internal data structures, which effectively hides the set of excluded key-value pairs when performing prefix-, range- or normal lookups.

Snapshots are identified via descriptive names, which are provided by users at creation time. If no name is provided, a timestamp reflecting the current server time is used. To trigger the creation of a snapshot, the snapshot tool sends a respective request to the MRC via the XtreemFS client. The request contains the name, if provided, and the directory for the snapshot. In turn, the MRC creates an instantaneous snapshot of

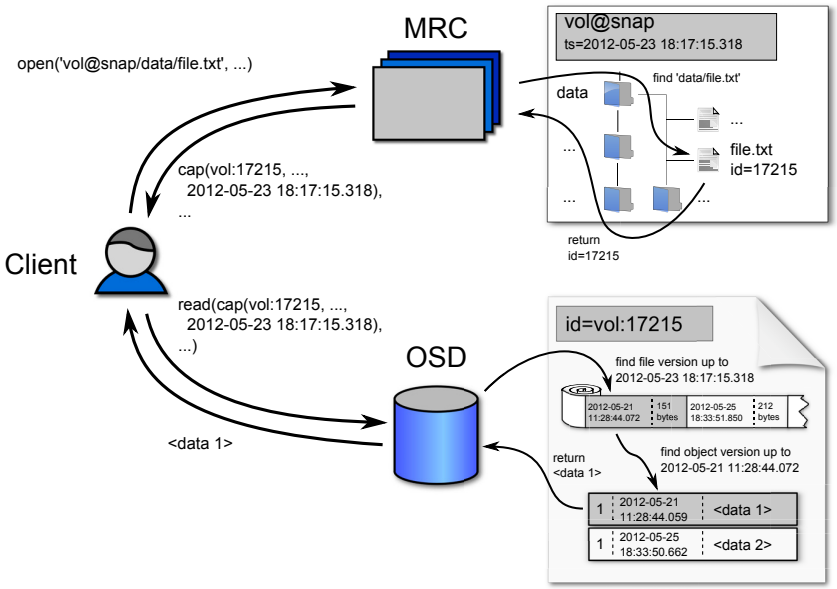

Figure 6.1: Illustration of interactions and internal procedures when accessing a file on a snapshot

the respective volume database in BabuDB and records a key-value pair comprising name and timestamp in a separate snapshot management database.

6.1.2 Accessing Snapshots

Figure 6.1 gives an overview of the individual processing steps that are triggered when accessing a file on a snapshot. To open a file, the client sends an `open` request containing a volume name followed by a snapshot name (`vol@snap`) along with a path to the file on the snapshot (`data/file.txt`).

When receiving such an `open` request, the MRC selects the snapshot `snap` of the volume database `vol` and resolves the path `data/file.txt` in order to retrieve the metadata of `file.txt`. If the request is successfully authorized, it responds with a capability and a list of all replicas containing information about the OSDs holding the file content. Aside from the file ID `vol:17215`, the capability contains the timestamp `2012-05-23 18:17:15.318`[1] attached to the database snapshot `vol@snap`, which is retrieved from the snapshot management database.

An OSD that receives a timestamped capability along with a read request searches its local file version log for the latest file version up to the timestamp, as described in

[1] In fact, timestamps are 64-bit integers reflecting the milliseconds elapsed since 1970.

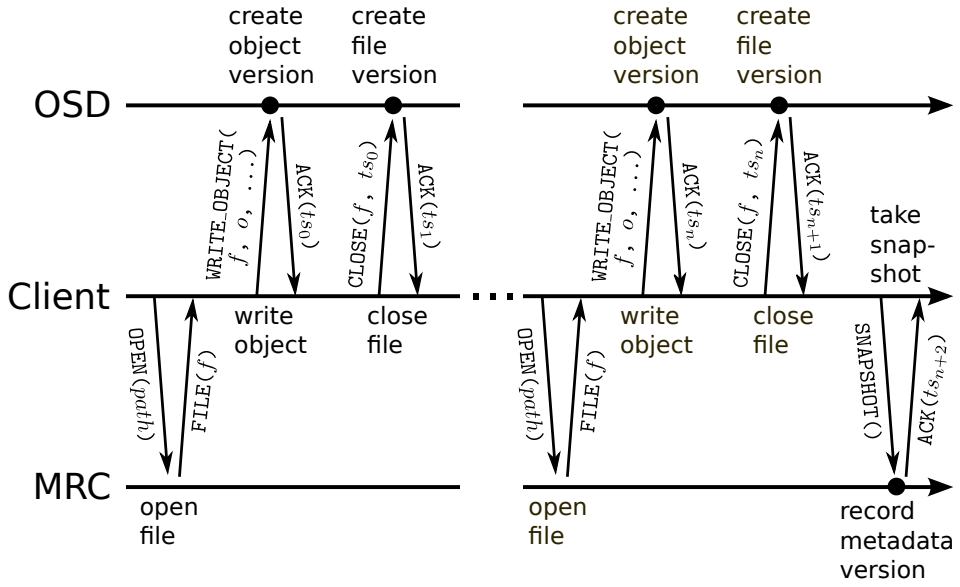

Figure 6.2: Illustration of redundantly created versions. Before taking a snapshot, the client repeatedly overwrites an object of a file and closes the file, thus enforcing the creation of multiple new file and object versions. However, only the newest file and object version will be bound to the snapshot; the other ones will never be accessed and are hence redundant.

sections 3.3 and 5.4. It scans its local object version table for the object version with the largest timestamp up to the file version timestamp and returns the requested data from this version.

6.2 Version Cleanup and Snapshot Deletion

File and object versions are created with every `write` or `close` operation, which may cause them to add up to considerable numbers over time. However, they are only of practical relevance if they are actually bound to a metadata version. File and object versions may become redundant, as they will never be accessed because they are superseded by later versions created prior to the next snapshot. The same may happen if metadata versions are removed in the course of deleting a snapshot. Figure 6.2 shows an example that illustrates the incurrence of redundant file and object versions.

6.2.1 Version Cleanup Algorithm

To free the additional space occupied by redundant object versions and to reduce the size and management overhead of file version logs and object version tables, a cleanup process can be triggered on the OSD. The process scans all files for redundant file and object versions. In particular, it first cleans up the file version log and then removes all

object versions that are not bound to any of the remaining file versions. More specifically, it removes those object versions that are neither bound to any existing snapshot nor part of the file's current version.

Algorithm 6.1 shows the cleanup algorithm. A client initiating the cleanup process first fetches the version timestamps of all metadata versions $TS := \{ts : (ts, M) \in V\}$ from the MRC (lines 2-3, 6-7). With these timestamps, it triggers cleanup operations on all OSDs (line 4). For each file f, an OSD takes the following two steps (lines 9-10):

1. Determine and remove the set ${V_f}' \subseteq V_f$ of superseded file versions (lines 12-17). A file version $(ts, V) \in V_f$ supersedes a file version $(ts', V') \in V_f$ if $ts > ts'$ and no metadata version was created between the points in time at which (ts', V') and (ts, V) were created. Considering the fact that clocks are loosely synchronized and each timestamp $ts_m \in TS$ was assigned by the MRC whereas ts' and ts were assigned by the OSD, this translates to the following condition: $\nexists ts_m \in TS\ (ts' - \epsilon' \leq ts_m \leq ts + \epsilon')$. ${V_f}'$ can be determined by checking this condition for any two file versions in V_f.

2. Determine and remove the set ${O_f}' \subseteq O_f$ of superseded object versions (lines 18-24). This step is similar to the former one, except that the timestamps from the remaining set of file versions $F := \{ts : (ts, l) \in V_f\}$ are used. Since file and object versions were timestamped with the same OSD clock, however, no grace period needs to be considered when comparing timestamps.

To ensure correctness even if new snapshots are created concurrently with cleanup runs, it is sufficient to add a timestamp reflecting the MRC's current time to the set of snapshot timestamps TS. Accordingly, the cleanup process on an OSD would only affect those file versions with timestamps that are least ϵ' before this timestamp.

The overall time and resource consumption of a cleanup run grows linearly with the number of snapshots, files, file versions and object versions. Accordingly, cleanup runs can have a negative effect on the performance of a large-scale file system. However, they can be performed in a fully asynchronous fashion, which implies that they can be scheduled at times of low utilization and suspended and resumed at any time.

6.2.2 Deleting Snapshots

Snapshots can be deleted fast, although they may cover large numbers of file versions stored across numerous OSDs. From a user's perspective, it is essentially sufficient to delete the metadata snapshot on the MRC, as this makes it impossible to list or open any file versions attached to the snapshot. In addition to the removal of the respective database snapshot along with its physical on-disk representation, this involves a removal of the timestamp from the volume's version management index. Once this timestamp has been removed, a cleanup run can be performed in order to asynchronously dispose of all file and object versions that were exclusively attached to the deleted snapshot.

Algorithm 6.1: Version cleanup

Client

triggers a cleanup run

```
1  procedure cleanup()
2      send FETCH_TIMESTAMPS() to MRC                retrieve metadata timestamps
3      receive TIMESTAMPS(TS) from MRC               wait for response
4      send CLEANUP(TS) to OSD                       trigger cleanup run
```

MRC

retrieves all metadata version timestamps

```
5  on FETCH_TIMESTAMPS()
6      TS ← {ts : (ts, M) ∈ V}                       determine all version timestamps
7      send TIMESTAMPS(TS) to Client                 send back timestamps
```

OSD

performs a cleanup run with a set of metadata timestamps TS

```
8  on CLEANUP(TS)
9      foreach local file f do                       for each local file ...
10         cleanup(f, TS)                            ... perform an individual cleanup
```

removes superseded file and object versions of f for a given set of metadata version timestamps TS

```
11 procedure cleanup(f, TS)
12     V_f' ← ∅                                      initialize set of superseded file versions
13     foreach (ts, l) ∈ V_f do
14         foreach (ts', l') ∈ V_f do
15             if ts < ts' ∧ ∄ts_m ∈ TS : (ts − ε' ≤ ts_m ≤ ts' + ε') then
16                 V_f' ← V_f' ∪ {(ts, l)}           record superseded file version
17     V_f ← V_f \ V_f'                              remove superseded file versions
18     F ← {ts : (ts, l) ∈ V_f}                      retrieve remaining file version timestamps
19     O_f' ← ∅                                      initialize set of superseded object versions
20     foreach (o, ts, d) ∈ O_f do
21         foreach (o', ts', l') ∈ O_f do
22             if o = o' ∧ ts < ts' ∧ ∄ts_f ∈ F : (ts ≤ ts_f ≤ ts') then
23                 O_f' ← O_f' ∪ {(o, ts, d)}        record superseded object version
24     O_f ← O_f \ O_f'                              remove superseded object versions
```

6.3 Atomic Modifications

Certain operations appear to be executed in a single step from a user's perspective, but in fact, they involve multiple interactions between the client and one or more servers. A common example is a `write` operation that exceeds the size of a single object and hence needs to be split up into multiple object writes. Such object writes can be executed by the client in parallel and may even be directed to different OSDs if the file is striped. Similarly, `truncate` operations on a striped file may require the creation or removal of objects across different hosts.

Snapshots may fail to capture such modifications in an atomic fashion if versions are individually timestamped, regardless of whether their creation was induced by an atomic modification. If a snapshot is concurrently taken with an atomic modification, it may reflect some but not all of its constituent updates. This is not desirable, as it may cause the snapshot to contain files in inconsistent states. The problem can be alleviated by exclusively creating versions on close, as this effectively excludes operations on open files like `write` operations from the set of relevant atomic modifications. On the other hand, it still requires the `close` operation to be executed atomically.

6.3.1 Recording Atomic Modifications

An approach to circumvent the problem is to enforce the assignment of a single timestamp to all versions created in the course of an atomic modification. This can, e.g., be done by attaching a timestamp ts to all constituent updates, which is known to be close to the servers' current local clock times. Such a timestamp ts can either be calculated from recently received server timestamps or immediately fetched from a server. A server receiving such a timestamped update, in turn, attaches ts to the newly recorded version instead of a timestamp generated by its local clock. This effectively ensures that all resulting versions appear to have been created concurrently according to definition 4.13 and will hence be reflected by a snapshot either in their entirety or not at all.

However, such an approach involves that individual changes may be dated to times in the future or past. The former induces delays when executing the respective requests on the servers, whereas the latter may cause these changes to appear on snapshots that were created longer ago than ϵ'. The decision whether or not atomic modifications shall be captured in an atomic fashion is thus a matter of choice, which we leave to the user.

6.3.2 Atomic File Size Updates

Writes and truncates that only affect a single object may also qualify as atomic modifications if it is essential to keep file sizes in the metadata consistent after every write. Besides being defined by the file's objects stored across the OSDs, the file size is also part of the file's metadata, which is returned by the MRC in response to a `stat` request. It is crucial to guarantee some degree of consistency between the file size in the metadata and the actual size of the file content, as many applications rely on the metadata file size to determine the end of a file. XtreemFS uses an asynchronous file size

update protocol to keep these file sizes in sync [SKH+08]. When a `write` or `truncate` operation changes the effective size of the file content on an OSD, the OSD responds with its new file size. The client reports new file sizes back to the MRC in regular intervals until the file is closed, which eventually causes metadata and content file sizes to become consistent.

Since metadata is versioned and timestamped in response to snapshot requests, it is not feasible to enforce a common timestamp on file content and metadata updates. Instead, the client contacts the respective OSDs rather than the MRC in order to retrieve the actual size of the respective file version when executing `stat` on a file in a snapshot.

6.4 Snapshots and Replication

A pivotal feature of XtreemFS is replication. Replication answers various purposes, which include (1) protection of data from loss as a result of hardware failures, (2) availability of data despite temporary or permanent outages of individual services and components, (3) load balancing to ensure that no bottlenecks occur when accessing popular chunks of data, and (4) locality of data to ensure that accessing close replicas provides high throughput and low latency.

XtreemFS supports replication of file content across OSDs, as well as metadata across MRCs and DIRs. The enforcement of a replica consistency model that is in line with the requirements of POSIX has turned out to be one of the main challenges in designing a replication scheme for XtreemFS. In particular, POSIX requires *strong consistency* for file system operations [Kol12]. This implies that all updates appear to be executed across all replicas in the same order, which corresponds to the real time order in which they were initiated. Barring concurrent reads and updates, this means that any read access to a replica must reflect the result of the latest previous update. If any replica of a file is read immediately after a replica of the same file has been successfully written by any process, the read operation must return the data resulting from the previous write operation.

6.4.1 Replication in XtreemFS

XtreemFS guarantees strong replica consistency through replication protocols between the servers. Different such protocols exist, as there are different types of replication in XtreemFS:

- **Read-only file replication**. Read-only replication has been designed to provide for a fast and efficient access to globally distributed write-once data. When closing a read-only replicated file after the initial write procedure, the file becomes immutable, and data is transmitted to the remaining replicas in an asynchronous, best-effort manner. Consistency is ensured when reading a replica of the file. If a client attempts to read an object that has not yet been received, it is fetched from a remote replica and stored locally before a response is sent to the client. Since

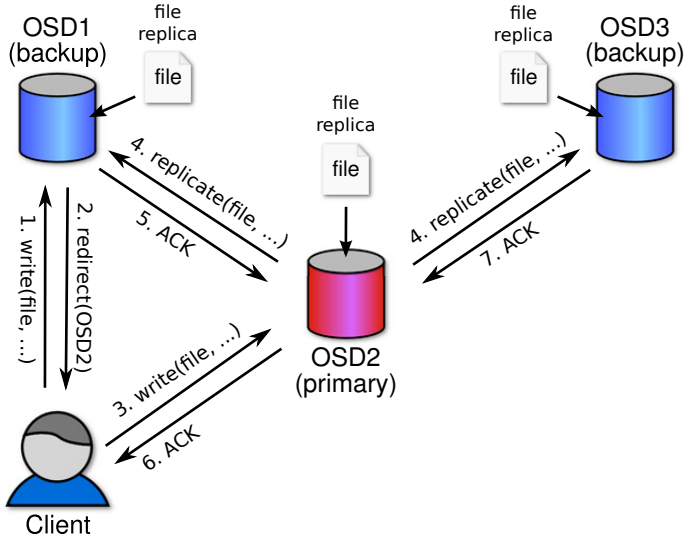

Figure 6.3: Primary-backup replication in XtreemFS. Requests directed to backup replicas are redirected to the current primary. The primary enforces a total order on all updates and forwards them to the backup replicas. With the "write quorum, read quorum" approach, any `write` call can be acknowledged to the client as soon as two of the three replicas have been successfully updated.

read-only replication involves write-once files, it conflicts with the principle of versioning and will thus not be taken into further consideration.

- **Read-write file replication**. Read-write replication is a more generic replication scheme for arbitrary files. It ensures consistency through a primary-backup [BMST93] mechanism. As shown in figure 6.3, all client accesses are directed to a designated primary for the replica. The primary disseminates any updates received from a client to the remaining backup replicas in the order in which they were received and thus enforces a total order upon all updates. The "write all, read any" approach reports a `write` call to the client as having been performed successfully after having received acknowledgments from *all* replicas, while it allows to read the current version of all data on *any* replica. The "write quorum, read quorum" approach only requires a *majority* of replicas to respond before acknowledging the `write` call. As a result, updates may be missed by a minority of replicas, which makes it necessary to read also from a majority of replicas. To render the read procedure more efficient, however, reads are restricted to the primary instead. In turn, it must be ensured that the primary is always aware of the latest version, which is done by means of a *replica reset* [Kol12] phase.

  To retain system availability if the primary becomes unavailable, the primary role is bound to a lease, which is negotiated between all servers in a decentralized manner [KHSH11] based on a variant of the Paxos [Lam98] algorithm. When the

lease times out without being renewed, the primary role is implicitly revoked, and any former backup replica can become the new primary.

- **Metadata replication**. In accordance with the separate management of file content and metadata in XtreemFS, metadata is replicated independently. Similar to read-write file replication, metadata replication is based upon a primary-backup scheme with leases for primary fail-over. It is implemented at database level in BabuDB; any update of a database record on the primary is sent to all or a majority of remote replicas, while read access is restricted to the primary.

When accessing a replicated file, the client receives a list of replicas from the MRC in response to an `open` call. With any subsequent `read`, `write` or metadata-related call, it attempts to access all replicas one by one until a response is received within a given time frame. Such a response can either be an acknowledgment confirming the successful execution of the operation, or a redirect message that refers the client to the current primary. This approach guarantees fault tolerance if individual replicas of a file are unavailable.

6.4.2 Replicating Snapshots

To preserve the properties of snapshots in the face of accesses to different replicas, it is necessary to also replicate information about versions of files and metadata. In particular, it must be ensured that the consistency of replicas guarantees the consistency of snapshots.

Consistency of Replicated Snapshots. Replicating snapshots in XtreemFS requires file replicas to be aware of file versions. For this purpose, any update received in the course of data dissemination between replicas could be processed by its recipient in a similar manner as an update received from a client. Accordingly, the versioning infrastructure would trigger the creation of new locally timestamped object or file versions if necessary.

However, if file versions of different replicas of a file receive different individual timestamps from their local servers, reading files in a snapshot may result in reading different versions of the file content, depending on the replica that is accessed. More specifically, the problem of snapshot consistency in a replicated setup can be considered as a special case of the problem of recording atomic modifications discussed in section 6.3, where each update of a replicated file corresponds to an atomic modification that affects all replicas. Thus, consistency is guaranteed by communicating and storing the timestamp that was initially generated on the primary along with all updates received and executed on backup replicas.

Bring metadata replication together with snapshots does not require any specific enhancements, as all metadata of a volume is centrally managed by a single MRC and replicated at database level. All updates including requests to capture database snapshots are internally disseminated and processed by the backup replicas. Timestamps of

metadata versions are stored in a separate index of the database, thus ensuring that a replicated MRC acts like a replicated state machine.

Replica Reset Phase. When a replica becomes primary, it must ensure that its local state is up-to-date, as it may have missed updates in the past. For this purpose, it queries all backup replicas for their latest locally known states. Once a majority has responded, it updates its local data by fetching missing updates if necessary. This procedure is referred to as the *replica reset* [Kol12] phase and needs to be executed when a read-write replicated file is opened, a replicated DIR or MRC is started, or a primary fail-over is required because the former primary becomes unavailable.

With read-write replicated files, it is necessary to also fetch the set of file versions that were missed by the new primary. More specifically, it is necessary to transmit all previously missed object versions in addition to the latest missed ones, along with all missed entries in the file version log.

Replica Set Changes. The set of replicas of a file may change over time, as file replicas can be added and removed at any time. Read-write replicated files are kept consistent in the face of replica set changes through a specific *replica set modification protocol* [Kol12], which ensures that a majority of replicas has a consistent view on the new replica set version as well as the file content. The latter is done by enforcing a *replica reset* on a majority of replicas in the new set, which in turn ensures that the majority receives information about all file and object versions created in the past.

Replica sets are managed by the MRC, as they are part of a file's metadata. However, changing the current replica set does not affect the metadata of previous file versions. Metadata versions defining previous snapshots may thus contain references to unavailable OSDs that have been removed in the meantime. In the worst case, none of the OSDs referenced by a former metadata version may be available anymore, which makes it impossible to determine which OSDs to contact when reading files on a snapshot.

To solve the problem, the MRC needs to keep track of the latest replica set change of each file. Each time a replica set change takes place, it adds a corresponding key-value pair to a specific *replica set index* in its database, which comprises the file ID as a key and the new replica set as a value. We use a separate index in addition to the main index for the metadata in order to ensure that the information remains accessible when the file is deleted. When opening a file on a snapshot, a lookup on the index retrieves an updated version of the replica set if available.

7 Evaluation

This chapter presents various experiments that demonstrate the characteristics of the snapshot algorithm and its implementation in XtreemFS. Our experiments evaluate the performance characteristics of metadata management (7.1), followed by an evaluation of file versioning and copy-on-write (7.2).

7.1 Metadata Management

We performed different experiments to evaluate the metadata management infrastructure. Our experiments address

- the performance of BabuDB as an MRC back-end, compared to the performance of BerkeleyDB for Java and ext4;
- a comparison of BabuDB with BerkeleyDB for Java and ext4, based on real-world traces;
- the impact of asynchronously written checkpoints on file creation performance;
- the additional latency of metadata operations experienced when accessing snapshots.

7.1.1 BabuDB Performance

Our first experiment evaluates the performance of the MRC in creating and accessing metadata of files. We used an XtreemFS volume with a single directory in order to measure the duration of file creations and directory listings.

To quantify the advantage of BabuDB for the management of file metadata over traditional approaches, we implemented two alternative MRC storage back-ends. The first one is based on *BerkeleyDB for Java 3.3.82*, a key-value store based on B-trees. To provide for a fair comparison with BabuDB, we disabled locking and transactions and enabled deferred writes. The second one is backed by the *ext4* file system, which internally resorts to *Htrees*. Htrees are a special variation of B-trees, which are optimized for file system workloads; they have a high fanout and do not need to be rebalanced. For each file created in XtreemFS, the ext4 back-end creates a corresponding empty file on a local ext4 file system.

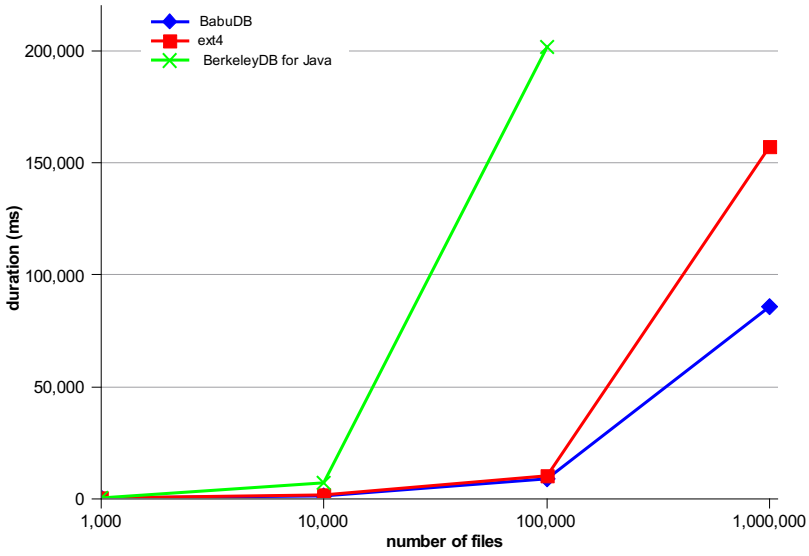

Figure 7.1: Duration of file creations

Setup. We performed the experiment on a single machine with a 4-core 2 GHz CPU, 4 GB of RAM and two SAS 10k RPM hard drives. To populate the directory, we initially executed batches of file creation requests of varying sizes between 1,000 to 1,000,000 on the MRC. Once all files were created, we executed an `ls` operation on the directory, which internally caused a `readdir` operation to be executed, followed by a `stat` operation on each individual file. We measured the total duration of the `create` and `ls` operations with and without the influence of system caches. To disable caching, we forced BabuDB to create a checkpoint, forced all updates to be synchronously written to disk, and dropped all system caches between creating the files and executing the `ls`. We conducted the experiment once with each of the three storage back-ends.

Results. As shown in figure 7.1, the ext4 and BabuDB back-end exhibit a similar performance for up to 100,000 file creations, but BabuDB scales better with larger directories. BerkeleyDB for Java has a good performance for very small databases but does not scale well. Creating 1,000,000 files was not possible with BerkeleyDB for Java due to timeouts in the client.

The performance characteristics shown for `ls` in figure 7.2 are similar. For very large directories, BabuDB is even faster when caches were flushed i.e., when reading from disk. This can be attributed to the fact that Java's red-black trees, which are used for the in-memory trees in BabuDB, are not optimized for range reads.

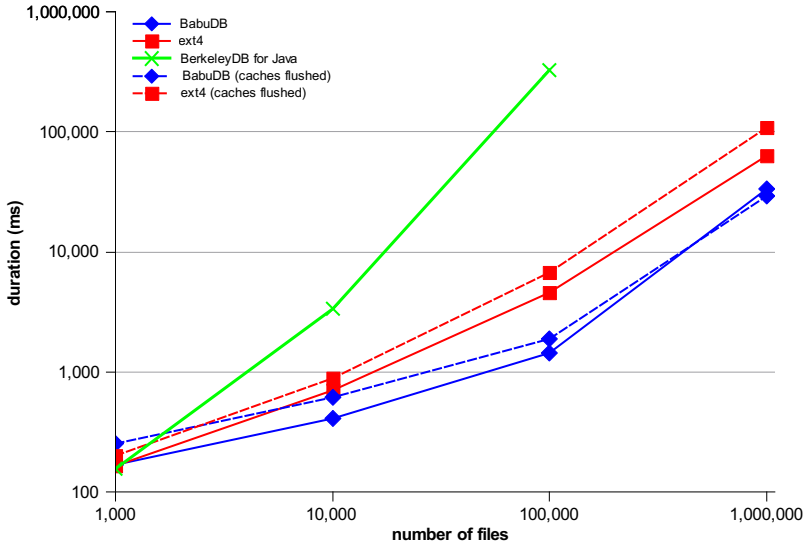

Figure 7.2: Duration of `ls` operation (with and without caches)

7.1.2 Trace-based Performance Analysis

To examine the performance of BabuDB in comparison with the other storage back-ends in a practical scenario, we performed two experiments in which we recorded and replayed traces of a Linux kernel build and a mail server stress test. We used the same physical machine to perform the experiments as for the BabuDB performance evaluation in section 7.1.1.

Linux Kernel Build. We used a FUSE module to record all metadata operations executed while building the Linux kernel (2.6.27). We replayed the trace sequentially with the BabuDB, the ext4 and the BerkeleyDB for Java back-end and measured the total execution time. The trace consists of 9.9 million operations (44% `stat`, 40% `open`, 15% `readlink`, 1% others).

While replaying the traces on the BabuDB and the ext4 back-ends took similar times (1,799 and 1,904 seconds, respectively), BerkeleyDB for Java needed 36,323 seconds to complete. We analyzed a partial replay of the trace with a profiler. The profiling revealed that BerkeleyDB for Java still does subtree locking despite locking being disabled in the database configuration. However, locking only accounted for approximately 50% of the time, while the rest of the time was spent on the B-tree search, insert and remove operations as well as the persistent operations log.

Dovecot IMAP Server. We used the same FUSE client as for the Linux kernel build test to record the metadata operations executed by a Dovecot mail server in the mailbox directory. We configured Dovecot to use the `maildir` format and to be compatible with

networked file systems, which, for instance, implies that no `mmap` or `fcntl` locking was used. To generate load, we ran the `imaptest` stress test provided by the Dovecot developers, which simulates a workload generated by five concurrent clients for 10 minutes. The resulting trace consisted of approximately 2 million metadata operations (51% `stat`, 48% `open`, 1% others).

Again, BabuDB and ext4 showed a similar performance with 367 and 385 seconds, respectively. BerkeleyDB for Java was an order of magnitude slower with 5,912 seconds.

7.1.3 Asynchronous Checkpoints

In a third experiment, we evaluated the impact of asynchronously written checkpoints on the performance of BabuDB and the MRC.

Setup. We used the same single-machine setup as we used for the previous experiments. We performed tests with 100,000 and 1,000,000 file creation requests, which we sequentially executed on the MRC, once without any checkpointing and once while concurrently writing database checkpoints. We configured BabuDB to use one of the two physical disks to store its internal operations log, while using the other one to store the database checkpoint files. We measured the latency of each individual file creation request.

Results. Table 7.1 shows the results, which demonstrate that BabuDB can provide acceptable response rates even while writing checkpoints to disk. The high standard deviation σ along with the low P_{99} for 1,000,000 files with concurrently written checkpoints indicate that there are some outliers with very high latency; we determined a maximum of 1.5 ms. However, the vast majority of the requests (P_{99}) showed only an increase of up to 30% in the response time while writing a snapshot.

# files	*avg. [ms]*	*σ [ms]*	*P_{99} [ms]*	*size*
normal				
100,000	0.1908	1.9145	0.1974	16 MB
1,000,000	0.1934	3.2928	0.2047	155 MB
during checkpoint				
100,000	0.3166	4.6740	0.2563	16 MB
1,000,000	0.4155	15.4259	0.2251	155 MB

Table 7.1: Duration of a file creation with and without concurrent asynchronous checkpointing. σ is the standard deviation, P_{99} the maximum among the fastest 99% of all requests, size is the on-disk index size.

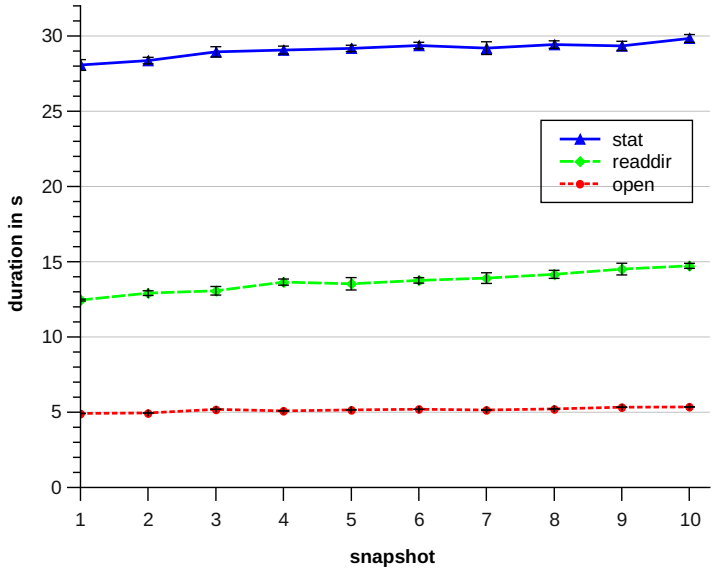

Figure 7.3: Access latency of metadata snapshots. The error bars show the standard deviation of the results of 10 test runs.

7.1.4 Metadata Snapshots

We conducted another experiment to determine the latency experienced when accessing snapshots of file system metadata stored in BabuDB, depending on the number of existing snapshots.

Setup. The test was performed on a local cluster connected by a gigabit Ethernet. Each cluster node is equipped with an 8-core 2.3 GHz CPU, 8 GB of RAM and a local hard drive. We set up an XtreemFS installation with one DIR, MRC and OSD and one single-threaded client on different cluster nodes. We disabled database checkpointing on the MRC in order to omit unforeseeable performance impacts on individual operations.

To assess the latency impact of snapshots on metadata accesses, we populated a volume with 10,000 files, which we randomly distributed across a tree of 500 directories with an average depth of 4. We repeated the procedure of taking a snapshot, performing a test run, deleting the tree, and creating the same tree again 10 times. Thus, we created a total of 10 snapshots, each having the same number of files and directories arranged in the exact same directory structure. With each test run, we measured the duration of 10,000 *open*, *readdir* and *stat* operations on the previously created snapshot. We performed an initial test run before any snapshots were created, so as to determine the latency when accessing the current version without any snapshots. We repeated each test run 10 times.

Results. The results are shown in figure 7.3. The graphs for the three different operation types show a slight increase in latency with a growing number of snapshots, which is traceable to the additional management overhead of BabuDB. While the increase is rather small for `open` and `stat` operations (6.3% and 8.7% for the first 10 snapshots, respectively), it is larger with `readdir` (18.6%); however, this can be attributed to the fact that each `readdir` operation had to retrieve the metadata of 20 files on average, rather than that of a single file. Moreover, the number of internal index structures BabuDB has to merge when executing a prefix lookup in connection with a `readdir` operation grows linearly with the number of snapshots in memory, which also contributes to a larger overhead.

Initial test runs on the current version show a similar performance impact for accesses to the first snapshot for `open` (5.05 seconds, σ = 0.2 seconds) and `readdir` (12.42 seconds, σ = 0.19 seconds). For `stat`, latency was substantially smaller (4.24 seconds, σ = 0.06 seconds, as compared to 28.07 seconds, σ = 0.36 seconds). The reason is that no additional file size lookup on the OSD is needed, which adds the latency of a communication round-trip when snapshots are enabled.

7.1.5 Summary

The experiments show that BabuDB is well-suited for file system metadata storage. It has a similar performance as ext4 but scales better with large numbers of files. The trace-based analysis confirms that BabuDB performs well with real-world workloads, which is further supported by the fact that asynchronous checkpoints have a low impact on the latency of operations. We have also shown that the number of metadata snapshots has no major impact on the latency of metadata accesses.

7.2 File Versioning

The algorithm and implementation described in the previous chapters captures snapshots without any communication between servers. This comes at the cost of a versioning scheme that creates potentially redundant file versions prior to the actual snapshot operation. To assess the viability of this approach, it is necessary to investigate the extent to which the versioning scheme affects the file system. In particular, we conducted three different experiments in order to determine

- the impact of versioning on the throughput attained when writing files;
- the impact of versioning and version retrieval on the throughput attained when reading snapshots of a file;
- the cost of versioning in terms of additional latency and storage consumption on a common file system workload.

All tests were run on the same hardware as the metadata snapshot experiment described in section 7.1.4.

7.2.1 Write Throughput

The first experiment gives information on the impact of file content versioning and COW on write throughput. To assess the maximum overhead and performance impact, we created a worst-case scenario with a "version on every write" policy, which ensured that each write operation induced the creation of a new object and file version.

Setup. We set up an XtreemFS installation consisting of one DIR, one MRC, one OSD, and 10 individual clients, each running on a separate cluster node. To prepare the experiments, we used the clients to create a total of 1,000 files on the OSD, each of these only consisting of a single object. We used this setup in order to maximize the number of file versions created and hence the potential overhead caused by the versioning. Once all files were created, the 10 clients repeatedly performed write operations for a duration of 60 seconds on randomly selected files. We used multiple clients in order to ensure that the OSD was permanently busy and no idle times occurred. With each write operation, `object size - 1` bytes were written at offset `0`, so that the complete object except for the last byte was replaced. We skipped the last byte to trigger the COW mechanism; otherwise, an internal optimization in the OSD would have been effective that would have created new object versions without prior copying. We measured the aggregated number of write operations performed across all clients and counted the total number of object versions (and file versions, respectively) created across all OSDs.

We conducted the experiment with varying object sizes. To quantify the cost of COW and versioning, we repeated the whole set of experiments twice, once with and once without versioning enabled. We used the average values of 10 test runs for our evaluation.

Results. Figure 7.4 shows the effective write throughput, as well as the number of file and object versions created for different object sizes. The effective write throughput was calculated by means of the formula $throughput = \frac{op\_count \times write\_size}{duration}$, where op_count refers to the measured number of completed operations, $write\_size$ to the number of bytes written with each operation, and $duration$ to the 60 second duration of a test run.

A general observation from the upper throughput diagram is that small object sizes come with reduced throughput rates, even if no versioning is enabled. With object sizes of up to 32 kB, throughput is CPU bound and effectively limited by the maximum number of requests an OSD can process during a certain time. With larger object sizes, throughput is I/O bound by the OSD's local hard disk. The diagram shows that typical object sizes of 128 kB or more reduce the total write throughput to no more than approximately 58% of the throughput seen without versioning.

Numbers from the lower version count diagram confirm that object version counts are roughly equal to the respective number of completed operations increased by the number of initially created file versions (i.e., 1000). Only small deviations of less than 0.1% occurred, as versions were also counted that were created through write operations that were still pending when a test run terminated after 60 seconds. The diagram

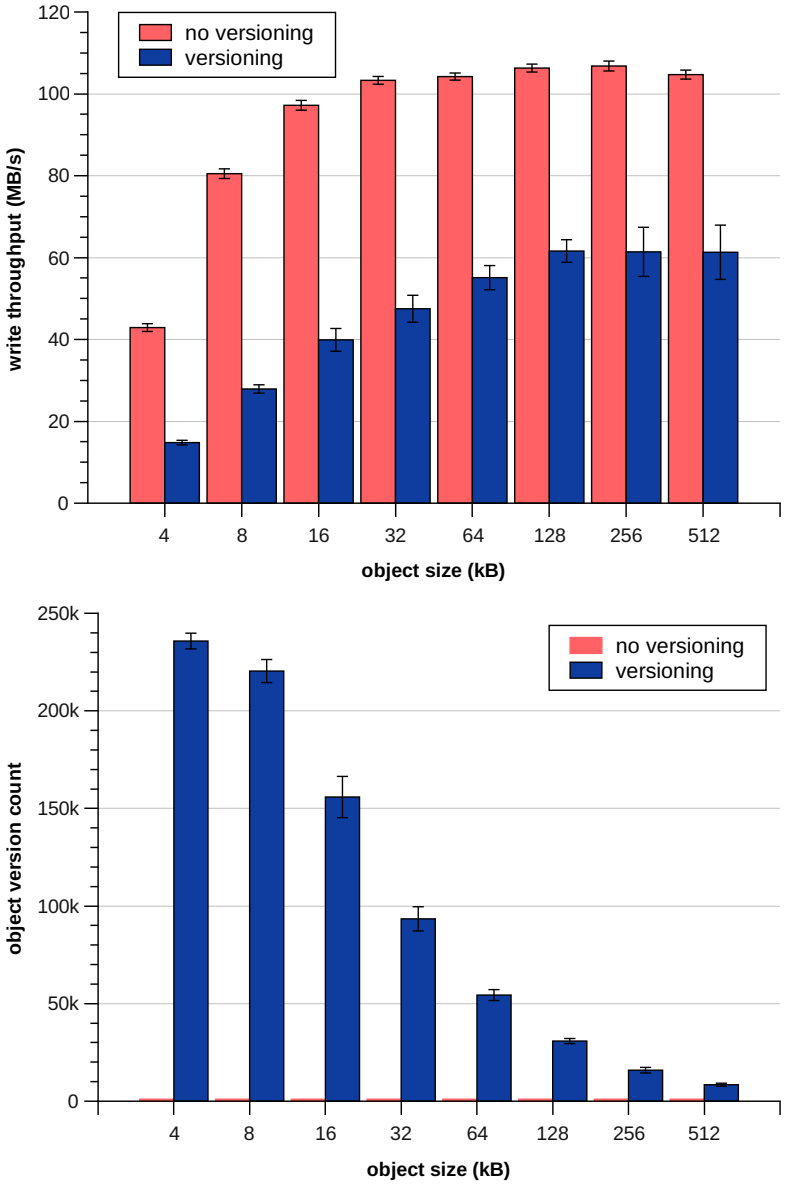

Figure 7.4: Aggregated write throughput and object version count for different object sizes. The error bars show the standard deviation of the results of 10 test runs.

also shows that large numbers of versions correlate with a reduced write throughput. Because of the increased management overhead induced by versioning, 4 kB writes attained only 34% of the throughput without versioning while generating approximately 236,000 versions, whereas 512 kB writes created only approximately 8,000 versions with a throughput of 59% of the throughput without versioning.

7.2.2 Read Throughput

The second experiment reveals the impact of file content versioning on read throughput. To assess the extent to which the management overhead of growing numbers of file and object versions affects the throughput when reading files, we created a set of files, on which we repeatedly created and accessed new versions.

Setup. We used the same setup as for the write throughput measurements. We populated an empty file system with 5,000 object-size files, which we repeatedly overwrote with a "version on every write" policy in order to trigger the creation of new versions. After having overwritten each file 10 times, we triggered a snapshot and repeatedly read randomly selected files in this snapshot for 60 seconds. Each time before reading files from a snapshot, we closed all files on the OSD and flushed all caches, so as to ensure that existing open states and cached data did not influence the results. We measured the aggregated number of read operations performed across all clients. We repeated this procedure 10 times, so that a total of 100 file content versions were created. After initially creating the files, we conducted a test run without versioning enabled in order to see the impact of versioning. We conducted the experiment with varying object sizes and ran each test 10 times.

Results. Figure 7.5 illustrates the outcome. Similar to the write throughput experiments described in the previous section, small object sizes come with a reduced overall performance, since the OSD is CPU-bound under these circumstances. However, the number of versions has no significant impact on the overall read performance. Even with 100 versions, the graphs show no evident decrease in the read throughput.

7.2.3 File System Workload

In a third experiment, we evaluated the overall performance impact and storage consumption of versioning on a common file system workload. We replayed a file system trace and measured the overall duration and storage consumption with and without versioning.

Setup. We used a similar setup as for the read and write throughput experiments, with one DIR, MRC and OSD running on different cluster nodes. Instead of running multiple clients to induce a high load on the OSD, however, we only set up a single client. We used the `dbench` file system benchmark tool to replay a file system trace

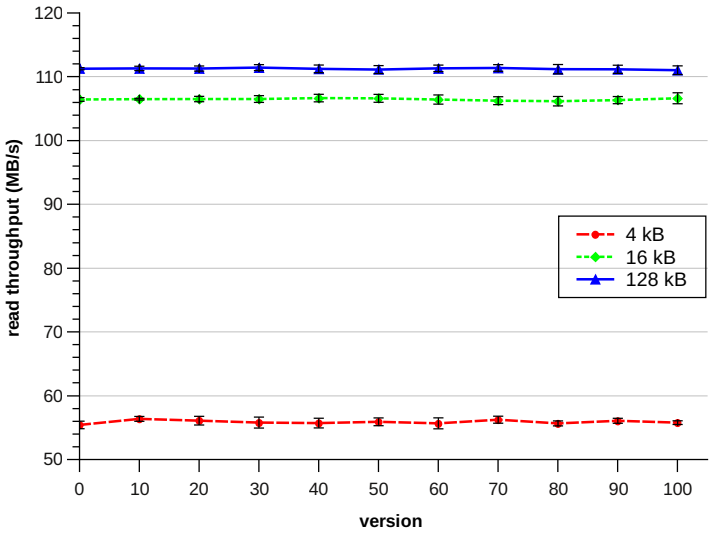

Figure 7.5: Aggregated read throughput for object sizes of 4 kB, 16 kB, and 128 kB. Version 0 represents the first test run without versioning enabled, all other versions are numbered according to their chronological order. The error bars show the standard deviation of the results of 10 test runs.

versioning	*duration [s]*	*σ [s]*	*# versions*	*occupied space*
disabled	2,246	11	1,049	79 MB
enabled	2,332	30	18,743	1,456 MB

Table 7.2: Results from a `dbench` run with approximately 460,000 operations. *Duration* is the average runtime of a test run, σ the standard deviation of the duration of 10 test runs, *# versions* the number of object versions created during the test run, and *occupied space* the total amount of occupied disk space after a test run.

reflecting the file system-related workload of a `NetBench` run[1]. More precisely, the trace contains a mixed workload of read, write and metadata accesses generated by a Windows client, which consists of approximately 460,000 operations (124k `read`, 97k `stat`, 80k `open`, 58k `close`, 40k `write`, 16k `unlink`, 45k others). We replayed the trace twice, once with and once without versioning enabled, and measured the total duration as well as the resulting storage consumption of each replay. We repeated the experiment 10 times with an object size of 128 kB and a "version on close" versioning scheme.

Results. The results are shown in table 7.2. The average duration was only about 4% higher with versioning enabled (2,332 seconds as compared to 2,246 seconds with-

[1] http://ftp.riken.go.jp/archives/pub/net/samba/unpacked/dbench/loadfiles/client.txt

out versioning), while the amount of generated object versions and consumed storage space was about 18 times higher. After a cleanup run, however, the total storage space could be reduced to the same amount as without versioning enabled.

7.2.4 Summary

The experiments have shown that versioning has a limited impact on the overall latency and throughput of read and write operations. While write throughput drops to roughly 60% of the maximum throughput in a worst-case scenario where a version is created with every write, versioning has no significant impact on read throughput. The replay of a real-world trace with a "version on close" policy has shown that the total runtime of a typical application workload does not significantly increase if versioning is enabled.

8 Conclusion

In this thesis, we have examined the problem of capturing and managing snapshots in large-scale distributed file system installations. Such installations comprise great numbers of hosts, which involves a high susceptibility to failures. They are subject to permanent access and must therefore be available at any time. Especially when running on a global scale, they are connected through wide-area networks with an increased communication latency.

To tackle the snapshot problem in the face of these determining factors and constraints, we have analyzed and discussed novel approaches to the traditional problems of capturing snapshots in *file systems* and *distributed systems*. The target systems for our approaches are file systems that follow the design principle of *object-based storage* [FMN+05, MGR03], which is present in various modern file systems like Lustre [Clu02] or Panasas Active Scale [TGZ+04].

8.1 Summary and Discussion of Results

The outcome is a description of a distributed algorithm that has been designed to capture and manage snapshots in scalable object-based file systems. To deal with the aforementioned challenges, the algorithm resorts to physical timestamps in order to capture a consistent snapshot of a file system stored across multiple servers, thus obviating the need for communication between these servers when taking the snapshot. Servers are assumed to have local hardware clocks, which are loosely synchronized such that the drift between any two clocks never exceeds ϵ. Based on a formal model, we have shown that such an upper bound of ϵ on the clock drift ensures that different file system servers effectively record their local shares of the global file system state within a time frame of approximately ϵ. On a wide area network, ϵ can typically be limited to a range of tens of milliseconds [Mil95].

We have also shown how to extend the algorithm such that it prevents anomalous behavior [Lam78] in the ordering of changes. Anomalous behavior can reverse the ordering in which different servers perceive and record updates to the file system. This may effectively disrupt the temporal ordering of file system operations stipulated by their causal dependencies. As a result, snapshots may reflect inconsistent global states that cannot have existed. Inspired by the approach followed in [ML04], we have shown that the problem can be solved by attaching timestamps to server messages combined with occasional short delays in the execution of operations, which, however, only can occur if clock drifts are effectively greater than message round-trip times.

We have provided a formal presentation of the algorithm, along with a description

of its implementation and practical challenges as part of XtreemFS [HCK+07], a scalable object-based file system. The algorithm records locally timestamped versions of file content on individual storage servers in response to specific local events, such as invocations of `write` or `close` operations on a file, and protects data from being overwritten by applying copy-on-write techniques at object level. It records a timestamped version of all metadata on the metadata server at database level when a file system snapshot is requested. File content and metadata versions are bound to each other through their timestamps, which effectively happens when reading files on a snapshot.

As a result, temporary downtimes and permanent crashes of individual storage servers do not inhibit the ability to capture snapshots, nor do they hamper access to parts of a snapshot on the remaining servers. Accordingly, the impact of failures is limited to the components on which they occur, which can further be reduced by employing replication techniques. Moreover, the synchrony of server clocks ensures that snapshots can be captured within a time frame of approximately ϵ, irrespective of the number of storage servers involved, which makes it possible to scale the system to any size. Our experiments have confirmed the practical suitability of our metadata management scheme and shown that versioning has no major impact a common file system workload in terms of latency and throughput.

8.2 Outlook and Future Work Perspectives

The algorithm described in this thesis has been implemented and published as part of the XtreemFS file system and thus been made available to a wide range of users. In the following, we point out directions and perspectives for future work in order to ease the administration of the snapshot infrastructure and to widen the range of its use cases.

8.2.1 Automated Clock Drift Determination

The consistency of snapshots depends upon the correct assessment of ϵ as an upper bound on the clock drift between servers. If the clock drift between two servers exceeds ϵ, typical effects are the inclusion of future changes into existing snapshots, anomalous behavior, as well as frequent and unexpectedly long delays when writing or closing files.

Currently, the assessment of ϵ has to be made by the administrator as part of the XtreemFS configuration, which implies that it is a static parameter based on manual observations and empirical values. A topic for future work is to keep track of the effective maximum clock drift in an automated and adaptive manner. Since common clock synchronization protocols like NTP [Mil91] essentially determine the drift of a local clock to its reference clock during the synchronization process, a possible approach is to report this drift to a central service such as the DIR, which keeps track of all individual clock drifts. In turn, servers could query the DIR in regular intervals and adjust their local values of ϵ accordingly.

8.2.2 Cleanup Scheduling

Aside from determining the upper bound on the clock drift, it is also up to the administrator to trigger cleanup runs on a regular basis, so as to minimize the additional storage consumption induced by versioning. By adjusting time and frequency of cleanup runs, it is effectively possible to trade off this additional storage consumption against the overall performance impact of cleanup runs.

A generic approach to ease the administrative overhead of cleanup runs while ensuring their minimal performance impact is to automatically schedule them at times of low activity on the file system. More precisely, OSDs could perform cleanup runs permanently in the background, which are automatically interrupted as soon as a predefined number of pending user requests is exceeded, and resumed when the user request count drops below the threshold again.

8.2.3 Continuous Data Protection

Our algorithm has been designed to capture snapshots of *entire* volumes or directories *on demand*. To provide the basis for continuous data protection, it is necessary to implement a versioning scheme that effectively creates a snapshot with *each* new version and provides access to the state of the file system at *any* time in the past, such as ext3cow [PB05a]. Such a versioning scheme would have to be more fine-grained, in that it maintains timestamped metadata versions of individual files. Accordingly, it would be necessary to build a separate versioning scheme on top of BabuDB instead of using its internal snapshot functionality, or to use a different data structure for the management of metadata, such as a versioned B-tree.

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 1–14, New York, NY, USA, 2002. ACM.

[ADD+08a] Nawab Ali, Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and P. Sadayappan. An OSD-based approach to managing directory operations in parallel file systems. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 175–184, September 2008.

[ADD+08b] Nawab Ali, Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and P. Sadayappan. Revisiting the metadata architecture of parallel file systems. In *Third Petascale Data Storage Workshop, Supercomputing*, November 2008.

[ADF+03] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *Proceedings of the 20th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 165–176, Washington, DC, USA, 2003. IEEE Computer Society.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[AFSM02] Alain Azagury, Michael E. Factor, Julian Satran, and W. Micka. Point-in-time copy: Yesterday, today and tomorrow. In *Proc. IEEE/NASA Conf. Mass Storage Systems (MSS)*, pages 259–270, 2002.

[AGGM04] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[AKM+07] Marcos K. Aguilera, Kimberly Keeton, Arif Merchant, Kiran-Kumar Muniswamy-Reddy, and Mustafa Uysal. Improving recoverability in multi-tier storage systems. *International Conference on Dependable Systems and Networks*, pages 677–686, 2007.

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li amd Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011: Fifth Biennial Conference on Innovative Data Systems Research*, pages 223–234, January 2011.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA, May 1995. ACM.

[BCD04] Fabián Bustamante, Brian Cornell, and Peter Dinda. Wayback: A user-level versioning file system for Linux. In *Proceedings of USENIX 2004 (Freenix Track)*, 2004.

[BGO+96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[BHS95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 277–288, Berkeley, CA, USA, 1995. USENIX Association.

[BHS09] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, March 2009.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BM07] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. http://opensolaris.org/os/community/zfs/docs/zfs last.pdf, 2007.

[BMLX03] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *MSST '03: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, Washington, DC, USA, 2003. IEEE Computer Society.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed*

systems (2nd Ed.), pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Bra98] Peter J. Braam. The Coda distributed file system. *Linux Journal*, 1998, June 1998.

[CAK+92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, Robert N. Sidebotham, and Transarc Corporation. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, 1999.

[CF07] Vineet Chadha and Renato J. Figueiredo. ROW-FS: a user-level virtualized redirect-on-write distributed file system for wide area applications. In *Proceedings of the 14th international conference on High performance computing*, HiPC'07, pages 21–34, Berlin, Heidelberg, 2007. Springer-Verlag.

[CIRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, October 2000.

[CJ91] Flaviu Cristian and Farnam Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 12–20, 1991.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[Clu02] Cluster File Systems, Inc. Lustre: a scalable, high-performance file system. Lustre Whitepaper Version 1.0, 2002.

[Cri89] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[DDWA07] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and Nawab Ali. Attribute storage design for object-based storage devices. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 263–268, September 2007.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34:375–408, September 2002.

[FB04] Michail Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block I/O level. In *21st IEEE Conference on Mass Storage Systems and Technologies*, pages 315–328, 2004.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FMN+05] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, pages 119–123, 2005.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[GGS06] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 269–277, New York, NY, USA, 2006. ACM.

[GGS10] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Efficient algorithms for global snapshots in large distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):620–630, May 2010.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amirit, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardint, Erik Riedelf, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, 1998.

[GNS88] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.

[Gob99] Howard B. Gobioff. *Security for a high performance commodity storage subsystem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1999. Chair-Gibson, Garth and Chair-Tygar, Doug.

[Gol85] Andrew C. Goldstein. Files-11 on-disk structure specification. VAX/VMS Software Development, 1985.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[Hél89] Jean-Michel Hélary. Observing global states of asynchronous distributed applications. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 124–135, London, UK, 1989. Springer-Verlag.

[HCK+07] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesús Malo, Jonathan Martí, and Eugenio Cesario. XtreemFS: a case for object-based storage in grid data management. In *3rd VLDB Workshop on Data Management in Grids*, 2007.

[HCK+08] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesús Malo, Jonathan Martí, and Eugenio Cesario. The XtreemFS architecture – a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20:2049–2060, December 2008.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, Berkeley, CA, USA, 1994. USENIX Association.

[HM03] R. J. Honicky and Ethan L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, April 2003. IEEE Computer Society.

[HM04] R. J. Honicky and Ethan L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004.

[HPR90] J.-M. Hélary, N. Plouzeau, and M. Raynal. Computing particular snapshots in distributed systems. In *IEEE Int Phoenix Conference on Computers and Communications*, March 1990.

[HWZ10] B. Halevy, B. Welch, and J. Zelenka. Object-Based Parallel NFS (pNFS) Operations. RFC 5664 (Proposed Standard), January 2010.

[IEE08] IEEE. IEEE Std 1003.1-2008. http://www.opengroup.org/onlinepubs/9699919799, 2008.

[Ins02] Institute of Electrical and Electronics Engineers, Inc. IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. IEEE Std. 1588, November 2002.

[JL96] James E. Johnson and William A. Laing. Overview of the Spiralog file system. *Digital Tech. J.*, 8(2):5–14, 1996.

[JLA00] Drew Roselli Jacob, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, 2000.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35:190–201, November 2000.

[KHSH11] Björn Kolbeck, Mikael Högqvist, Jan Stender, and Felix Hupfeld. Flease - lease coordination without a lock server. In *IPDPS*, pages 978–988. IEEE, 2011.

[KK03] Muniswamy-Reddy Kiran-Kumar. VERSIONFS: A versitile and user-oriented versioning file system. Master's thesis, Stony Brook University, December 2003.

[Kol12] Björn Kolbeck. *A Fault-Tolerant and Scalable Protocol for Replication in Distributed File Systems*. PhD thesis, Humboldt University Berlin, April 2012.

[KRS95] Ajay D. Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4):224–233, 1995.

[Ksh10] Ajay D. Kshemkalyani. Fast and message-efficient global snapshot algorithms for large-scale distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 21:1281–1289, September 2010.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lis93] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

[LMJ07] Andrew W. Leung, Ethan L. Miller, and Stephanie Jones. Scalable security for petascale parallel file systems. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[LRV87] Hon Fung Li, Thiruvengadam Radhakrishnan, and K. Venkatesh. Global state detection in non-FIFO networks. In *ICDCS*, pages 364–370, 1987.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:382–401, July 1982.

[LTT96] Edward K. Lee, Chandramohan A. Thekkath, and Ramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MAC+08] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 2008 EuroSys Conference*, pages 41–54, April 2008.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[McC90] Kirby McCoy. *VMS File System Internals*. Digital Press, 1990.

[MGR03] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 8:84–90, 2003.

[Mil91] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[Mil95] David L. Mills. Improved algorithms for synchronizing computer network clocks. *IEEE/ACM Transactions on Networking*, 3:245–254, 1995.

[Mil03] David L. Mills. A brief history of NTP time: Memoirs of an internet timekeeper. *SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

[ML04] Chuang-Hue Moh and Barbara Liskov. TimeLine: A high performance archive for a distributed object store. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 351–364, 2004.

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36:31–44, December 2002.

[MOW98] Peter Muth, Patrick O'Neil, and Gerhard Weikum. Implementation and performance of the LHAM log-structured history data access method. In *Proceedings of the 24th VLDB Conference*, pages 452–463, 1998.

[MRH09] Kiran-Kumar Muniswamy-Reddy and David A. Holland. Causality-based versioning. In *FAST '09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 15–28, 2009.

[MRWHZ04] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association.

[NCI04] T10 Technical Committee NCITS. SCSI object-based storage device commands -2 (OSD-2). Project T10/1721-D, Revision 0, October 2004.

[NF96] Nuno Neves and W. Kent Fuchs. Using time to improve the performance of coordinated checkpointing. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, pages 282–291, Washington, DC, USA, 1996. IEEE Computer Society.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[ODCH+85] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Michael Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. Technical Report UCB/CSD-85-230, EECS Department, University of California, Berkeley, Apr 1985.

[OM05] Christopher Olson and Ethan L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 64–73, New York, NY, USA, 2005. ACM.

[PB05a] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[PB05b] Kristal T. Pollack and Scott A. Brandt. Efficient access control for distributed hierarchical file systems. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 253–260, Washington, DC, USA, 2005. IEEE Computer Society.

[PLG+07] Kristal Pollack, Darrell D. E. Long, Richard Golding, Ralph Becker-Szendy, and Benjamin C. Reed. Quota enforcement for high-performance distributed storage systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 72–84, September 2007.

[PV02] Attila Pásztor and Darryl Veitch. PC based precision timing without GPS. *SIGMETRICS Perform. Eval. Rev.*, 30(1):1–10, 2002.

[QD02] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *FAST '02: Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[Qui91] Sean Quinlan. A cached WORM file system. *Software - Practice and Experience*, 21(12):1289–1299, 1991.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1991.

[RS93] Parameswaran Ramanathan and Kang G. Shin. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Transactions on Software Engineering*, 19:571–583, June 1993.

[RV10] Julien Ridoux and Darryl Veitch. Principles of robust timing over the Internet. *Communications of the ACM*, 53(5):54–61, 2010.

[SCB08] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[SEN10] S. Shepler, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Minor Version 1 Protocol. RFC 5661 (Proposed Standard), January 2010.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP '99)*, volume 33, pages 110–123, New York, NY, USA, December 1999. ACM.

[SFHV99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7. IEEE Computer Society, 1999.

[SGSG03] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 231–244, January 2002.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[SKH+08] Jan Stender, Björn Kolbeck, Felix Hupfeld, Eugenio Cesario, Erich Focht, Matthias Hess, Jesús Malo, and Jonathan Martí. Striping without sacrifices: Maintaining POSIX semantics in a parallel file system. In *LASCO'08: First USENIX Workshop on Large-Scale Computing*, 2008.

[SKHH10] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. BabuDB: Fast and efficient file system metadata storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pages 51–58, Washington, DC, USA, 2010. IEEE Computer Society.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *ACM SIGOPS Operating Systems Review*, 36:15–30, December 2002.

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), April 2004. Updated by RFCs 3980, 4850, 5048.

[SRO96] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The global file system. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage (MSST'96)*, pages 319–342, 1996.

[ST84] Amitabh Shah and Sam Toueg. Distributed snapshots in spite of failures. Technical Report TR84-624, Cornell University, Computer Science Department, July 1984.

[SX05] Liuba Shrira and Hao Xu. SNAP: Efficient snapshots for back-in-time execution. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 434–445, Washington, DC, USA, 2005. IEEE Computer Society.

[SX06] Liuba Shrira and Hao Xu. Thresher: An efficient storage manager for copy-on-write snapshots. In *Proceedings of the Usenix Annual Technical Conference*, Boston, MA, May 2006.

[Tay89] Kimberly E. Taylor. The role of inhibition in asynchronous consistent-cut protocols. In *Workshop on Distributed Algorithms*, pages 280–291, 1989.

[TBM+11] Andy Twigg, Andrew Byde, Grzegorz Miłoś, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified B-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, 2011.

[TGZ+04] Hong Tang, Aziz Gulbeden, Jingyu Zhou, William Strathearn, Tao Yang, and Lingkun Chu. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, page 53, November 2004.

[TKT92] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. Rollback recovery in distributed systems using loosely synchronized clocks. *IEEE Transactions on Parallel and Distributed Systems*, 3:246–251, March 1992.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[Tre05] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ACM Computing Research Repository*, 501002, 2005.

[UAA+10] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 225–238, 2010.

[Ven89] S. Venkatesan. Message-optimal incremental snapshots. In *ICDCS*, pages 53–60, 1989.

[vHvSAvMT11] Richard van Heuven van Staereling, Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Efficient, modular metadata management with Loris. In *Sixth International Conference on Networking, Architecture, and Storage (NAS 2011)*, pages 278–287. IEEE Computer Society, 2011.

[Wan06] Feng Wang. *Storage Management in Large Distributed Object-Based Storage Systems*. PhD thesis, University of California, Santa Cruz, December 2006.

[WB07] Joel C. Wu and Scott A. Brandt. Providing quality of service support in object-based file system. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 157–170, Washington, DC, USA, 2007. IEEE Computer Society.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, November 2006.

[WBML04] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. OBFS: A file system for object-based storage devices. In *MSST '04: Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Techonlogies*, pages 283–300, 2004.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.

[WBW96] Christopher Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the server for the Spiralog file system. *Digital Technical Journal*, 8:15–31, January 1996.

[Wei04] Sage A. Weil. Leveraging intra-object locality with EBOFS. UCSC CMPS-290S project report, University of California, Santa Cruz, May 2004.

[WPBM04a] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004. IEEE Computer Society.

[WPBM04b] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Intelligent metadata management for a petabyte-scale file system. In *Intelligent Storage Workshop*, Minneapolis, MN, USA, May 2004.

[XLY+06] Weijun Xiao, Yinan Liu, Qing (Ken) Yang, Jin Ren, and Changsheng Xie. Implementation and performance evaluation of two snapshot methods on iSCSI target storages. In *Proceedings of NASA/IEEE Conference on Mass Storage Systems and Technologies*, 2006.

[XYR+09] Weijun Xiao, Qing (Ken) Yang, Jin Ren, Changsheng Xie, and Huaiyang Li. Design and analysis of block-level snapshots for data protection and recovery. *IEEE Transactions on Computers*, 58:1615–1625, December 2009.

List of Figures

List of Tables

List of Algorithms

Selbständigkeitserklärung

Ich erkläre, dass

- ich die vorliegende Arbeit mit dem Titel “Snapshots in Large-Scale Distributed File Systems” selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin vom 17.01.2005, zuletzt geändert am 13.02.2006, bekannt ist, gemäß Amtl. Mitteilungsblatt Nr. 34/2006.

Berlin, den 13. September 2012

Jan Stender